VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PRATIK JANA(1BM22CS356)

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 **PRATIK JANA (1BM22CS356),** 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.

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

Index

Github Link: BIS-LAB/ at main · pratik03092003/BIS-LAB

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

Implement A genetic Algorithm to MAximize
A fn ($f(x)=x^2$)

```
import numpy np
def objective-fn (x)
        return (x**2)
population_size=100
mutation_size=0.01
crossover_rate = 0.7
num generation = 50
xmin = -10
xmax = 10

def intialize (size)
        return np.random uniform (x-min
                                  x-max
                                  size)
def evaluate (population)
        return objective-fn (population)

def select (population, fitness)
        selected_indius= np.random.choice
                         (len(population),
        size = 2, replace - False
        return population [selected_indius[
                np.argmax (fitness-indiu)
```

```
def crossover (parent, parent2)
    if np.random.rand() < crossover
                                rate
        return (parent + parent2)/2
    return parent 1

def mutate (individual):
        if np.random.rand() < mutation
                                rate
        return Individual + np.random
            uniform (-1,1)
        return Individual

def genetic_algorithm():
        population = Initialize (population_size)
        best_soln = none
        best_fitness = -np.inf
return best fitness . best_soln
```

Best x

o/p = 9.96203495

Best value 99.25534684

Code:

```
import numpy as np

# Define the objective function
def objective_function(x):
    return x**2

# Initialize parameters
population_size = 100
mutation_rate = 0.01
crossover_rate = 0.7
num_generations = 50
x_min = -10
x_max = 10

# Create initial population
def initialize_population(size):
    return np.random.uniform(x_min, x_max, size)

# Evaluate fitness
def evaluate_fitness(population):
    return objective_function(population)

# Selection (Tournament Selection)
def select(population, fitness):
    selected_indices = np.random.choice(len(poTpulation), size=2, replace=False)
    return population[selected_indices[np.argmax(fitness[selected_indices])]]

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

# Mutation
def mutate(individual):
    if np.random.rand() < mutation_rate:
        return individual + np.random.uniform(-1, 1)  # Random mutation
    return individual

# Genetic Algorithm
def genetic_algorithm():
    population = initialize_population(population_size)
    best_solution = None
    best_fitness = -np.inf

    for generation in range(num_generations):
```

```
        fitness = evaluate_fitness(population)

        # Track the best solution
        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]

        # Create a new population
        new_population = []
        for _ in range(population_size):
            parent1 = select(population, fitness)
            parent2 = select(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring)
            new_population.append(offspring)

        population = np.array(new_population)

    return best_solution, best_fitness

# Run the Genetic Algorithm
best_x, best_value = genetic_algorithm()
print(f"Best x: {best_x}, Maximum value of f(x): {best_value}")
```

```
Best x: 9.96260349526031, Maximum value of f(x): 99.25346840377296
```

Program 2

Particle Swarm Optimization for Function Optimization

Algorithm:

```
import numpy as np
import matplot.lib

def objective fn (x):
        return np.sum(x)*2+x)

Initialize parameter
  np, nD, nI, c1, c2 = 30, 2, 20, 0.5

Initialize global value
global_best_position = personal_best_position
                                (personal_best
global_best value = np.min (      -value)
                              personal_best_val)

plt.figure (figsize=(10,8))
plt.xlim (-10,10)
plt.ylim (-10,10)
plt.xlabel (x axis)
plt.ylabel (y axis)
plt.title ("PSO")
```

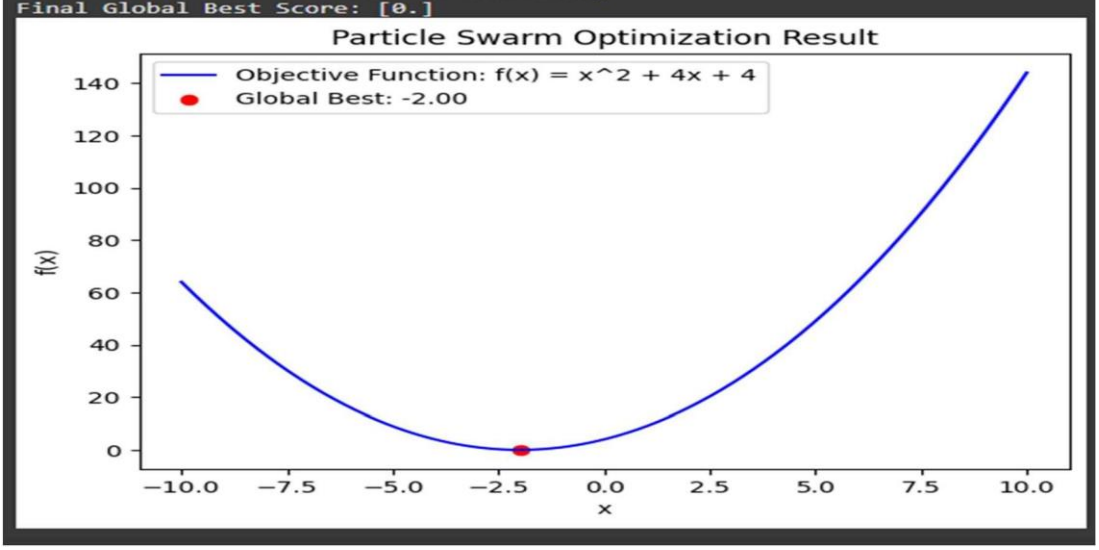
```
for i in range (num_particles)
    r1,r2 = np.random.rand(2)
    inertia = inertia_weight * particles_velocity[i]
    cognitive = c1*r1*(personal_best_pos[i] -
                                particle_pos[i])
    social = c2*r2*(gbp - pp[i])
    particles_velocity[i] = inertia + cognitive
                                    + social
    pp[i] += particles_velocity[i]

# Evaluate fitness

    fitness = objective_fn(particles_position[i])
    if fitness < personal_best_value[i]:
        personal_best_value[i] = fitness
        personal_best_position[i] = particles_
                                    position[i]

    if fitness < global_best_value:
        global_best_value = fitness
        global_best_position = particles_pos[i]

plt.show()
print (global_best_position)
print (global_best_value)
```

Code:

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

# Objective function: f(x) = x^2 + 4x + 4
def objective_function(x):
    return x**2 + 4*x + 4

# PSO parameters
num_particles = 30        # Number of particles
dimensions = 1            # Problem dimensionality (1D for this example)
iterations = 100          # Number of iterations
w = 0.5                   # Inertia weight
c1 = 1.5                  # Cognitive coefficient
c2 = 1.5                  # Social coefficient

# Initialize the particles
positions = np.random.uniform(-10, 10, size=(num_particles, dimensions))  # Random positions
velocities = np.random.uniform(-1, 1, size=(num_particles, dimensions))  # Random velocities
personal_best_positions = np.copy(positions)  # Personal best positions
personal_best_scores = np.array([objective_function(p) for p in positions])  # Personal best scores

# Global best (initially the best personal position)
global_best_position = personal_best_positions[np.argmin(personal_best_scores)]
global_best_score = np.min(personal_best_scores)

# PSO Optimization loop
for iteration in range(iterations):
    for i in range(num_particles):
        # Update velocity
        r1 = np.random.rand()
        r2 = np.random.rand()
        velocities[i] = w * velocities[i] + c1 * r1 * (personal_best_positions[i] - positions[i]) + c2 * r2 *
(global_best_position - positions[i])

        # Update position
        positions[i] = positions[i] + velocities[i]

        # Evaluate the objective function
        current_score = objective_function(positions[i])

        # Update personal best
        if current_score < personal_best_scores[i]:
            personal_best_scores[i]  =  current_score
            personal_best_positions[i] = positions[i]

        # Update global best
```

```
        if current_score < global_best_score:
            global_best_score = current_score
            global_best_position = positions[i]

    # Optionally print the global best score during the iterations
    if iteration % 10 == 0:
        print(f"Iteration {iteration}: Global Best Score = {global_best_score}")

# Final result
print(f"Final Global Best Position: {global_best_position}")
print(f"Final Global Best Score: {global_best_score}")

# Plotting the results for visualization
x = np.linspace(-10, 10, 400)
y = objective_function(x)

plt.plot(x, y, label="Objective Function: f(x) = x^2 + 4x + 4", color='blue')
plt.scatter(global_best_position, global_best_score, color='red', label=f"Global Best:
{global_best_position[0]:.2f}")
plt.legend()
plt.title("Particle Swarm Optimization Result")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.show()
```

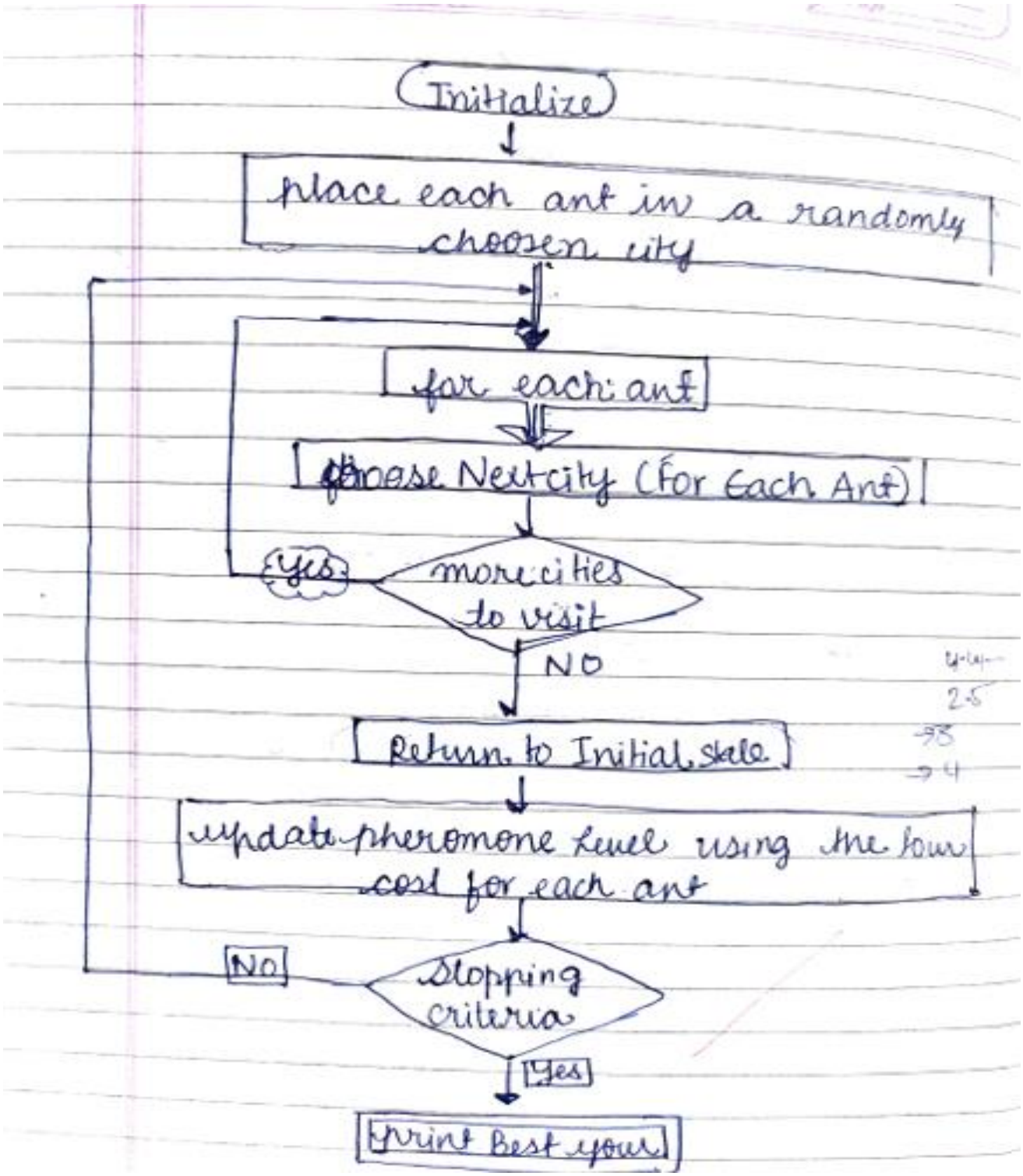
```
Iteration 0: Global Best Score = [0.00063526]
Iteration 10: Global Best Score = [0.00020327]
Iteration 20: Global Best Score = [5.5574043e-06]
Iteration 30: Global Best Score = [1.19586296e-09]
Iteration 40: Global Best Score = [2.02327044e-11]
Iteration 50: Global Best Score = [1.72306613e-13]
Iteration 60: Global Best Score = [0.]
Iteration 70: Global Best Score = [0.]
Iteration 80: Global Best Score = [0.]
Iteration 90: Global Best Score = [0.]
Final Global Best Position: [-1.99999999]
Final Global Best Score: [0.]
```



Program 3

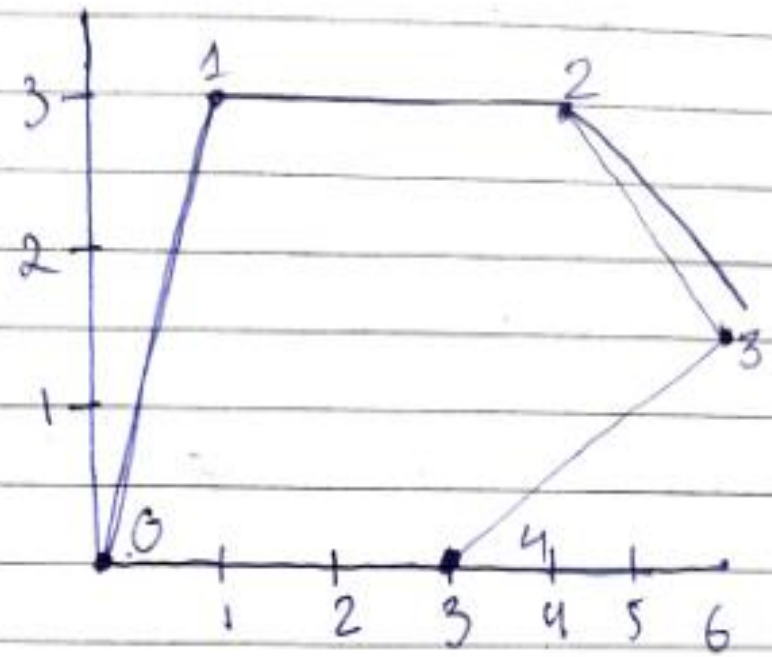
Ant Colony Optimization for the Traveling Salesman Problem

Algorithm:



```
import numpy as np
import random
import matplotlib.pyplot as plt

cities = np.array([[0,0],[1,3],[4,3],[6,1],[3,0]])
num_ants, num_iteration = 5,100
alpha, beta, rho, initial_pheromone = 1.0, 2.0, 0.1, 0.0

def choose_next_city (current_city, visited):
        probs= [(pheromones [current_city][j]
                                ** alpha)
            * (1 / distance [current city][j] ** beta)


Algorithm:

best_route, best_distance = None, float ('inf')
        routs, distance = [],[]
        for in range (num_iteration):
                routes, distance_list = [], [ ]
        for _ in range (num_ants):
                route = [random.randint
        while len(route) < num cities (0, num_cities-1)]

            d = sum[distance [route[i]]
                                route [i+1]]
```

```
routes.append(route)
distance_list.append(distance)

# update pheromones
# plotting.
```



Best route ACO : 15:15 unit

Code:

```
import random
import math
import numpy as np

# Calculate the Euclidean distance between two cities
def distance(city1, city2):
    return math.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)

# Ant Colony Optimization for TSP
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  # Influence of pheromone
        self.beta = beta     # Influence of distance
        self.rho = rho       # Pheromone evaporation rate
        self.iterations = iterations
        self.pheromone = np.ones((self.num_cities, self.num_cities))  # Initial pheromone
        self.distances = np.zeros((self.num_cities, self.num_cities))

        # Calculate the distance matrix for all pairs of cities
        for i in range(self.num_cities):
            for j in range(i + 1, self.num_cities):
                self.distances[i][j] = distance(self.cities[i], self.cities[j])
                self.distances[j][i] = self.distances[i][j]

    def probability(self, ant, city, visited):
        """Calculates the probability of moving to a next city."""
        pheromone = self.pheromone[city]
        heuristic = np.array([1.0 / self.distances[city][i] if i not in visited else 0 for i in
range(self.num_cities)])
        pheromone_heuristic = pheromone ** self.alpha * heuristic ** self.beta
        pheromone_heuristic[visited] = 0  # Ensure no city is visited twice

        return pheromone_heuristic / pheromone_heuristic.sum()

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

        # Iterate for a number of iterations
        for _ in range(self.iterations):
            all_tours = []
            all_distances = []
```

```
        # Each ant constructs a solution
        for ant in range(self.num_ants):
            visited = [0]  # Start from city 0
            tour = [0]
            total_distance = 0

            # Construct the solution by visiting all cities
            while len(visited) < self.num_cities:
                city = visited[-1]
                prob = self.probability(ant, city, visited)
                next_city = np.random.choice(range(self.num_cities), p=prob)
                visited.append(next_city)
                tour.append(next_city)
                total_distance += self.distances[city][next_city]

            # Add the return to the starting city
            total_distance += self.distances[visited[-1]][visited[0]]

            # Track the best tour and distance
            all_tours.append(tour)
            all_distances.append(total_distance)

            if total_distance < best_distance:
                best_distance = total_distance
                best_tour = tour

        # Update pheromone trails
        self.pheromone *= (1 - self.rho)  # Evaporate pheromone
        for ant in range(self.num_ants):
            for i in range(self.num_cities - 1):
                city1 = all_tours[ant][i]
                city2 = all_tours[ant][i + 1]
                self.pheromone[city1][city2] += 1.0 / all_distances[ant]  # Pheromone reinforcement
                self.pheromone[city2][city1] += 1.0 / all_distances[ant]

    return best_tour, best_distance

# Example usage
if __name___== "__main__":
    # Define cities as a list of (x, y) coordinates
    cities = [(0, 0), (1, 3), (4, 3), (6, 1), (6, 5), (2, 7), (3, 4), (5, 2)]

    # Set the ACO parameters
    num_ants = 10
    alpha = 1.0    # Pheromone importance
    beta = 2.0     # Heuristic importance
    rho = 0.1      # Pheromone evaporation
```

```
iterations = 100

# Create and run the ant colony optimizer
aco = AntColony(cities, num_ants, alpha, beta, rho, iterations)
best_tour, best_distance = aco.run()

# Output the best solution found
print("Best tour:", best_tour)
print("Best distance:", best_distance)
```

```
Best tour: [0, 1, 6, 5, 4, 2, 7, 3]
Best distance: 24.772376032626727
```

Program 4

Cuckoo Search (CS)

Algorithm:

cuckoo search:

- import numpy as np
  from scipy.special import gamma,
  import random

// Rastrigin function

- def rastrigin(x)
  A=10
  return (A * len(x) + sum(x_i**2 - A*cos(2*pi*x_i)
  for (x_i) in x)

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

// create random sol

- def initialize_nest (num_nests, dim, lb, ub
  nests = np.random.uniform (Lb, up, (num_nests, dim))

  returns nest

- def levy_flight (x, alpha = 1.5, beta = 10, delta=
  sigma = (gamma(1+alpha) * np.sin(np.pi * alpha/2) /
  gamma((1+alpha)/2) * alpha * 2**((alpha-1)/2)))) ** (1/alpha)
  u = np.random.normal (0,1, len(x))
  v = np.random.uniform (0,sigma,len(x))

```
worst_nest_indices = np.argsort(fitness[-int
                    pa * num_nest):]
for idx in worst_nest_indices
nest[idx] = np.random.uniform(lb, ub, dim)
fitness[idx] = objective_fn(nest[idx])

current_best_idx = np.argmin(fitness)
current_best_fitness = fitness[current_best_idx]
if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
    best_nest = nests[current_best_idx]

    print("Iteration:" {iteration+1}/{num_iter
                                    ns}
          "best fitness:": {best_fitness})

    return best_nest, best_fitness
```

Code:

```
import numpy as np
import math

# Sphere Function: f(x) = sum(x_i^2)
def sphere_function(x):
    return np.sum(x**2)

# Lévy Flight function
def levy_flight(Lambda, d):
    # Lévy flight step size based on power-law distribution
    sigma_u = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = np.random.normal(0, sigma_u, d)
    v = np.random.normal(0, 1, d)
    step = u / np.abs(v)**(1 / Lambda)
    return step

# Initialize the Cuckoo Search Algorithm
def cuckoo_search(func, n_nests, n_dim, max_iter, pa=0.25, alpha=0.01, lambda_levy=1.5):
    # Initialize nests randomly
    nests = np.random.uniform(-5, 5, (n_nests, n_dim))  # Bound the values between -5 and 5 for the
Sphere function
    fitness = np.apply_along_axis(func, 1, nests)  # Calculate fitness of each nest

    # Keep track of the best solution found so far
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        # Generate new nests via Lévy flights
        new_nests = nests + alpha * levy_flight(lambda_levy, n_dim)
        # Ensure new nests are within bounds
        new_nests = np.clip(new_nests, -5, 5)

        # Evaluate new nests' fitness
        new_fitness = np.apply_along_axis(func, 1, new_nests)

        # Replace worst nests with new ones based on probability of discovery
        for i in range(n_nests):
            if np.random.rand() < pa:  # Discovery probability
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # Update the best solution if we find a better one
        if np.min(fitness) < best_fitness:
            best_fitness = np.min(fitness)
```

```
            best_nest = nests[np.argmin(fitness)]
        # Output the current iteration's best solution

    return best_nest, best_fitness

# Set algorithm parameters
n_nests = 50          # Number of nests (solutions)
n_dim = 10            # Dimensionality of the problem (number of variables)
max_iter = 100        # Number of iterations
pa = 0.25             # Probability of discovery (abandoning the worst nests)
alpha = 0.01          # Scaling factor for the Lévy flight
lambda_levy = 1.5     # Exponent for Lévy flight distribution

# Run the Cuckoo Search Algorithm to minimize the Sphere Function
best_solution, best_value = cuckoo_search(sphere_function, n_nests, n_dim, max_iter)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_value)
```

```
Best Solution Found: [ 0.02678509  0.23352996  1.6252229  -2.98750263  3.27095615 -2.01511818
  1.96138533  0.00816108  0.82788469 -0.18501841]
Best Fitness Value: 30.478578013430198
```

Program 5

Grey Wolf Optimizer (GWO)

Algorithm:

```
import numpy as np

def obj_fn(x)
    return np.sum(x**2)

def grey_w_o(obj_fn, dim, n_wolves=5
                max_itr=50, lb-10, ub+10):

    wolves - np.random.uniform(lb, ub, (n_wolves
                                        , dim))

    alpha = np.inf
    beta = np.inf
    delta = np.inf
    X_a = None
    X_b = None
    X_s = None

    for it in range(max_itr):
        fitness= np.array([objective_fn(wolf)
                    for wolf in values])

        sorted_index= np.argsor(fitness)
        X_a, X_b, X_b - wolves[sorted_index[:3]]
```

```
a = 2 - i * (2/maxitr)
new_wolves[]
for j in rang (n_wolves):
    r1, r2 = np.random.rand(dim),
        r2 = np.random.rand(dim),

    A1 = 2*a*r1 - a
    C1 = 2*r2
    D_alpha = abs(C1 * x_alpha - wolves[j])
    X1 = x_alpha - A1 * D_alpa

    r1, r2 = np.random.rand(dim),
             np.random.rand(dim)
    A2, C2 = 2*a*r1 - a, 2*r2
    D_beta = abs((2* x_b - wolves[j])
    x2 = X_beta - A2 * D_beta

    r1, r2 = np.random.rand(dim)
             np.random.rand(dim)
    A3, C3 = 2*a*r1 - a, 2*r2
    D_Delta = abs(C3 - X_delta - wolves[j])
    X3 = X_delta - A3 * D_Delta
    new_wolf = (x1+x2+x3) / 3
    new_wolves.append(np.clip(new_wolf, lb,
```

Code:

```
import numpy as np

# Define the objective function (Sphere function: sum(x^2))
def sphere(x):
    return np.sum(x**2)

# Grey Wolf Optimizer (GWO)
class GWO:
    def _init_(self, obj_func, dim, pop_size, max_iter, lb, ub):
        self.obj_func = obj_func  # Objective function to minimize
        self.dim = dim            # Number of dimensions
        self.pop_size = pop_size  # Number of wolves in the population
        self.max_iter = max_iter  # Maximum number of iterations
        self.lb = lb              # Lower bound of search space
        self.ub = ub              # Upper bound of search space

        # Initialize the wolves' positions randomly within bounds
        self.position = np.random.uniform(self.lb, self.ub, (self.pop_size, self.dim))
        self.fitness = np.array([self.obj_func(ind) for ind in self.position])  # Initial fitness of all wolves

        # Initialize the alpha, beta, and delta wolves' positions and fitness
        self.alpha_pos = np.zeros(self.dim)
        self.beta_pos = np.zeros(self.dim)
        self.delta_pos = np.zeros(self.dim)
        self.alpha_score = float('inf')  # Best score (we minimize, so start with infinity)
        self.beta_score = float('inf')
        self.delta_score = float('inf')

    def update_position(self, alpha, beta, delta, a, A, C, position):
        # Update the position of a single wolf based on the positions of alpha, beta, delta
        r1 = np.random.random(self.dim)
        r2 = np.random.random(self.dim)

        # Update position using the equation
        D_alpha = abs(C[0] * r1 - position - alpha)
        D_beta = abs(C[1] * r1 - position - beta)
        D_delta = abs(C[2] * r1 - position - delta)

        X1 = alpha - A[0] * D_alpha
        X2 = beta - A[1] * D_beta
        X3 = delta - A[2] * D_delta

        # New position is the average of the three components
        new_position = (X1 + X2 + X3) / 3
        return new_position
```

```
    def optimize(self):
        for t in range(self.max_iter):
            # Update parameters A and C based on the iteration
            a = 2 - t * (2 / self.max_iter)  # Declining over iterations
            A = np.random.uniform(-a, a, 3)
            C = np.random.uniform(0, 2, 3)

            # Evaluate fitness and update the alpha, beta, delta wolves
            for i in range(self.pop_size):
                fitness = self.obj_func(self.position[i])

                if fitness < self.alpha_score:
                    self.alpha_score = fitness
                    self.alpha_pos = self.position[i]

                elif fitness < self.beta_score:
                    self.beta_score = fitness
                    self.beta_pos = self.position[i]

                elif fitness < self.delta_score:
                    self.delta_score = fitness
                    self.delta_pos = self.position[i]

            # Update the position of each wolf in the population
            for i in range(self.pop_size):
                # Update the position of wolf i
                self.position[i] = self.update_position(self.alpha_pos, self.beta_pos, self.delta_pos, a, A, C,
self.position[i])

                # Ensure the new position stays within the bounds
                self.position[i] = np.clip(self.position[i], self.lb, self.ub)

            # Optionally print the progress
            print(f"Iteration {t+1}/{self.max_iter} - Best Fitness: {self.alpha_score}")

        return self.alpha_pos, self.alpha_score

# Set problem-specific parameters
dim = 10              # Number of dimensions (variables in the function)
pop_size = 50         # Number of wolves in the population
max_iter = 100        # Number of iterations
lb = -5.12            # Lower bound of search space
ub = 5.12             # Upper bound of search space

# Create an instance of the GWO class
gwo = GWO(obj_func=sphere, dim=dim, pop_size=pop_size, max_iter=max_iter, lb=lb, ub=ub)
```

```
# Run the optimization
best_pos, best_score = gwo.optimize()

print("\nOptimization Complete!")

print("Best Solution (Position):", best_pos)
print("Best Fitness (Value):", best_score)
```

```
Iteration 85/100 - Best Fitness: 0.010060709455679629
Iteration 86/100 - Best Fitness: 0.010060709455679629
Iteration 87/100 - Best Fitness: 0.010060709455679629
Iteration 88/100 - Best Fitness: 0.010060709455679629
Iteration 89/100 - Best Fitness: 0.010060709455679629
Iteration 90/100 - Best Fitness: 0.010060709455679629
Iteration 91/100 - Best Fitness: 0.010060709455679629
Iteration 92/100 - Best Fitness: 0.010060709455679629
Iteration 93/100 - Best Fitness: 0.010060709455679629
Iteration 94/100 - Best Fitness: 0.010060709455679629
Iteration 95/100 - Best Fitness: 0.010060709455679629
Iteration 96/100 - Best Fitness: 0.010060709455679629
Iteration 97/100 - Best Fitness: 0.010060709455679629
Iteration 98/100 - Best Fitness: 0.010060709455679629
Iteration 99/100 - Best Fitness: 0.010060709455679629
Iteration 100/100 - Best Fitness: 0.010060709455679629

Optimization Complete!
Best Solution (Position): [0.48660073 0.16603068 0.22455522 0.2916564  0.28708061 0.22164054
 0.45593904 0.29095408 0.33491953 0.21197124]
Best Fitness (Value): 0.010060709455679629
```

Program 6

Parallel Cellular Algorithms and Programs

Algorithm:

Lab8

(parallel cellular automation-based algorithm for optimal robot route planning grid-based environment, ensuring collision free navigation while minimizing travel distance and computational time

```
import random
import numpy as np

def initialize_grid (N,M):
np.random.choice ([0,1], size,([N,M]))

def count_live_neighbours (grid, i, j, N,M)
    return sum(
        grid[ni,nj] for ni in range (i-1, i+2)
        for nj in range (ni != i or nj != j)

def update_cell (grid, new_grid, i, j, N, M):
    Live_neighbours = count_live_neighbours
                (grid, i, j, N, M)
    if grid[i,j] == 1:
        new_grid [i, j] = 1 if live_neighbo
    else :
```

```
else:
    new_grid[i,j] = 1 if live_neighbours(grid,i,j,N,M)

def printgrid(grid)
    for row in grid :
        print(' '.join(map(str, row)))
    print()

def parallel game (N,M,steps)
    grid = initialize_grid(N,M)
        for i in range(steps)
            printgrid()
            for j in range (N)
        grid = new_grid
    print_grig(grid)

N,M = 5,5
steps = 5
final grid = 5
        = parallel of life(N,M,steps
```

Code:

```
import numpy as np
import random

# Objective function: f(x) = x^2 + 4x + 4
def objective_function(x):
    return x**2 + 4*x + 4

# Parameters
grid_size = 10              # Number of cells in the grid (1D grid here for simplicity)
num_iterations = 100        # Number of iterations
neighborhood_radius = 1     # Neighborhood range (cell's neighbors)
mutation_rate = 0.1         # Probability of mutation

# Initialize the grid: random values within a range (-10, 10)
def initialize_grid(grid_size):
    return np.random.uniform(-10, 10, grid_size)

# Fitness evaluation for each cell
def evaluate_fitness(grid):
    return np.array([objective_function(cell) for cell in grid])

# Update the state of each cell based on its neighbors
def update_states(grid, fitness, neighborhood_radius):
    new_grid = np.copy(grid)
    for i in range(grid_size):
        # Get the neighbors (with wraparound at boundaries)
        left = (i - neighborhood_radius) % grid_size
        right = (i + neighborhood_radius) % grid_size

        # Ensure that the indices are valid
        if left <= right:
            neighbors = grid[left:right+1]
            fitness_neighbors = fitness[left:right+1]
        else:
            # Handle wraparound correctly
            neighbors = np.concatenate([grid[left:], grid[:right+1]])
            fitness_neighbors = np.concatenate([fitness[left:], fitness[:right+1]])

        # Update rule: take the average of neighbors if their fitness is better
        best_neighbor = neighbors[np.argmin(fitness_neighbors)]
        # Update rule: Apply smaller mutation
        new_grid[i] = best_neighbor + random.uniform(-mutation_rate / 10, mutation_rate / 10) #
Reduced mutation impact

    return new_grid
```

```
# Main Cellular Algorithm Function
def parallel_cellular_algorithm():
    # Initialize grid
    grid = initialize_grid(grid_size)

    best_solution = None
    best_fitness = float('inf')

    # Iterate through generations
    for iteration in range(num_iterations):
        fitness = evaluate_fitness(grid)

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

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

        # Output the best solution at regular intervals
        if iteration % 10 == 0:
            print(f"Iteration {iteration}: Best Fitness = {best_fitness}")

    return best_solution, best_fitness

# Run the Parallel Cellular Algorithm
best_solution, best_fitness = parallel_cellular_algorithm()

# Output the final best solution
print(f"Final Best Solution: {best_solution}")
print(f"Final Best Fitness: {best_fitness}")
```

```
Iteration 0: Best Fitness = 0.008102765732833639
Iteration 10: Best Fitness = 0.0006178173275515064
Iteration 20: Best Fitness = 8.999472633774985e-08
Iteration 30: Best Fitness = 2.5049349261507814e-09
Iteration 40: Best Fitness = 2.5049349261507814e-09
Iteration 50: Best Fitness = 7.945333280190425e-11
Iteration 60: Best Fitness = 7.945333280190425e-11
Iteration 70: Best Fitness = 7.945333280190425e-11
Iteration 80: Best Fitness = 7.945333280190425e-11
Iteration 90: Best Fitness = 7.945333280190425e-11
Final Best Solution: -1.9999910863479302
Final Best Fitness: 7.945333280190425e-11
```

Program 7

Optimization via Gene Expression Algorithms

Algorithm:

Lab 9 :- BisLab:-

```
import random
# Def knapsack problem (objective fn)
def knapsnack problem (objective fn)
def knapsack_fitness (items, capacity, solution)
    total_wieght = sums[items[i][0] for i in
        range (len(solution) if solution[i]==1])
    total_value = sum[items[i][1] for i in range
        len(solution)) if solution[i]==1

# If total weight exceeds the capacity return
if total_wieght > capacity
        return 0
return total value

# gene expression algorithm (GEA)
    def_init_(self population size, numitems mut
        rate, crossover_rate, generation, cap
        items)
        self population_size = population size
        self num_items = num_items
        self mutation rate = mutation_rate
        self crossover rate = crossover_rate
        self capacity_rate = capacity
```

```
self population = []

# Initialize population with random sol (binary
representation

def initialize_population(self):

    self.population = [[random.randint(0,1) for _ in
    range(self.population_size)]

# Evaluate fitness of population
def evaluate_fitness(self):
    return [knapsack_fitness(self.items,
            self.capacity, individual) for
    individual in self.population

# Select individuals based on fitness (roulette wheel
                                        selection)

def selection(self):
    fitness_values = self.evaluate_fitness()
    total_fitness == 0: # Avoid division by zero
    return random.choices(self.population, k=
                          self.population_size)
```

```
# crossover.
def crossover(self, parent 1, parent 2):
    if random.random() < self.crossover_rate:
        crossover_point = random.randint(
                                1, self.num_items - 1)
    return individual


# evolve population over generation
def evolve(self):
    self.initialize_population()
    best_solution = None
    best_fitness = 0
    for gen in range(self.generations):
        # selection
            selected = self.selection()

# get user Input for gea parameter
def get_user_input
print ("Enter population size)
    best_solution, fitness = gea.evolve()
print(best solution)
```

Code:

```
import numpy as np
import random

# Objective function: f(x) = x^2 + 4x + 4
def objective_function(x):
    return x**2 + 4*x + 4

# GEA parameters
population_size = 30          # Number of genetic sequences (solutions)
num_genes = 1                 # Number of genes in each sequence (1D optimization in this case)
mutation_rate = 0.05          # Probability of mutation
crossover_rate = 0.7          # Probability of crossover
num_generations = 100         # Number of generations

# Initialize Population: Generate random genetic sequences (chromosomes)
def initialize_population(population_size, num_genes):
    return np.random.uniform(-10, 10, (population_size, num_genes))

# Fitness Evaluation: Evaluate fitness of each sequence
def evaluate_fitness(population):
    return np.array([objective_function(individual[0]) for individual in population])

# Selection: Tournament Selection
def select(population, fitness):
    selected_indices = np.random.choice(len(population), size=2, replace=False)
    return population[selected_indices[np.argmin(fitness[selected_indices])]]

# Crossover: Single-point crossover
def crossover(parent1, parent2):
```

```
    if random.random() < crossover_rate:
        # Random crossover point (for 1D, just average)
        return (parent1 + parent2) / 2
    return parent1

# Mutation: Introduce small random changes
def mutate(individual):
    if random.random() < mutation_rate:
        return individual + np.random.uniform(-1, 1)
    return individual

# Gene Expression: Translate genetic sequence to a functional solution
def gene_expression(population):
    return population

# Main GEA Function: Optimization Loop
def gene_expression_algorithm():
    # Initialize population
    population = initialize_population(population_size, num_genes)

    best_solution = None
    best_fitness = float('inf')

    # Track the best solution through generations
    for generation in range(num_generations):
        fitness = evaluate_fitness(population)

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

        # Create a new population
        new_population = []
        for _ in range(population_size):
            parent1 = select(population, fitness)
            parent2 = select(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring)
            new_population.append(offspring)

        population = np.array(new_population)

        # Gene expression (translation of genetic sequence to solutions)
        population = gene_expression(population)
```

```
        # Output the best solution at regular intervals
        if generation % 10 == 0:
            print(f"Generation {generation}: Best Fitness = {best_fitness}")
    return best_solution, best_fitness

# Run the GEA
best_solution, best_fitness = gene_expression_algorithm()

# Output final best solution
print(f"Final Best Solution: {best_solution}")
print(f"Final Best Fitness: {best_fitness}")
```

```
Generation 0: Best Fitness = 4.302998488681217
Generation 10: Best Fitness = 1.8683297042798586e-07
Generation 20: Best Fitness = 2.078337502098293e-13
Generation 30: Best Fitness = 0.0
Generation 40: Best Fitness = 0.0
Generation 50: Best Fitness = 0.0
Generation 60: Best Fitness = 0.0
Generation 70: Best Fitness = 0.0
Generation 80: Best Fitness = 0.0
Generation 90: Best Fitness = 0.0
Final Best Solution: [-2.00000001]
Final Best Fitness: 0.0
```