VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

S Gajanana Nayak (1BM22CS227)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **S Gajanana Nayak (1BM22CS227),** 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title	Page No.
1	18-10-2024	Genetic Algorithm for Optimization Problems	1-6
2	25-10-2024	Particle Swarm Optimization for Function Optimization	7-10
3	8-11-2024	Ant Colony Optimization for the Traveling Salesman Problem	11-15
4	15-11-2024	Cuckoo Search Optimization	16-20
5	22-11-2024	Grey Wolf Optimizer	21-26
6	29-11-2024	Parallel Cellular Algorithms and Programs	27-35
7	29-11-2024	Optimization via Gene Expression Algorithms	36-41

Github Link: https://github.com/Gajanana227/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:

LAB - 3 18/10/24

Genetic Algorithm:

Algorithm:

Step 1: Set population-size, mutation rate, crossover rate, num-generations, gene-length

Step 2: Generate population-size, individually each individual is random binary string of length (gene-length)

Step 3: Repeat step 2 for each generation (0 to no. of generations)

Step 4: Evaluate fitness of each generation individually

Step 5: Select parents based on fitness by performing roulette wheel selection

Step 6: Create new population based on
→ Crossover
→ Mutation
→ Replace old population

Step 7: Track the best fit solution

Step 8: Generate all the best solution.

Code:

```
import random
import numpy as np

def fitness-function(x):
    return x ** 2

population_size = 10
mutation-rate = 0.01
crossover-rate = 0.8
num-generations = 10
```

```
gene_length = 10
def create_population(size, gene_length):
    return [np.random.randint(0,2, gene_length).tolist() for
        _ in range(size)]

def binary_to_decimal(binary):
    binary_str = ''.join(str(bit) for bit in binary)
    return int(binary_str, 2) / (2** gene_length -1) * 20 -10

def evaluate_population(population):
    return [fitness_function(binary_to_decimal(individual))
            for individual in population]

def select(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [fitness / total_fitness for fitness in
                       fitness_scores]
    return population[np.random.choice(range(len(population)),
                      p = selection_probs)]
def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        crossover_point = random.randint(1, gene_length - 1)
        child1 = parent1[:crossover_point] + parent2[cross_over_pt:]
        child2 = parent2[:crossover_pt] + parent1[crossover_pt:]
        return [child1, child2]
    return [parent1, parent2]

def mutate(individual):
    for i in range(gene_length):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]

return individual
```

```
def genetic_algorithm():
    population = create_pop (pop-size, gene_length)
    for generation in range (num_gen):
        fitness_score = evaluate-pop(pop)
        best_fitness = max(fitness_score)
        best_individual = population [fitness_score.index(best)]
        print(f"Generation {gen}: Best Fitness = {best :.4f}")

        new-population = []
        while len (new_pop) < pop_size:
            parent1 = select (population, fitness scores)
            parent2 = select (population, fitness scores)
            offspring = crossover (parent1, parent2)

        population = new_pop [:pop-size]

    best_fitness = max (fitness_scores)
    best_individual = population [fitness_scores.index (best_fit)]
    best_solution = binary_to_decimal (best_indival)

    print(f"\n Best solution found: x = {best_solution :4
        f(x) = {fitness_function (best_solution):.4f}")

genetic_algorithm()
```

Output:

Generation 0: Best Fitness = 72.4912

Generation 1: Best Fitness = 92.3327

Generation 2: Best Fitness = 97.6677

Generation 3: Best Fitness = 92.3327

Generation 4: Best Fitness = 92.3327

Code:

```
import random

def fitness_function(x):
    return x ** 2

def generate_population(size, lower_bound, upper_bound):
    return [random.uniform(lower_bound, upper_bound) for _ in range(size)]

def selection(population, fitness_values):
    total_fitness = sum(fitness_values)
    probabilities = [f / total_fitness for f in fitness_values]
    return random.choices(population, weights=probabilities, k=len(population))

def crossover(parent1, parent2, crossover_rate):
    if random.random() < crossover_rate:
        alpha = random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    return parent1, parent2

def mutate(individual, mutation_rate, lower_bound, upper_bound):
    if random.random() < mutation_rate:
        individual += random.uniform(-1, 1)
        individual = max(lower_bound, min(upper_bound, individual))
    return individual

def genetic_algorithm(population_size, lower_bound, upper_bound, generations, mutation_rate,
crossover_rate):
    population = generate_population(population_size, lower_bound, upper_bound)
```

```
    for generation in range(generations):
        fitness_values = [fitness_function(ind) for ind in population]
        selected_population = selection(population, fitness_values)
        next_generation = []
        for i in range(0, len(selected_population), 2):
            parent1 = selected_population[i]
            parent2 = selected_population[i + 1 if i + 1 < len(selected_population) else 0]
            child1, child2 = crossover(parent1, parent2, crossover_rate)
            next_generation.extend([child1, child2])
        population = [mutate(ind, mutation_rate, lower_bound, upper_bound) for ind in next_generation]
        best_fitness = max(fitness_values)
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness:.4f}")
    return max(fitness_function(ind) for ind in population)

population_size = 10
lower_bound = -10
upper_bound = 10
generations = 9
mutation_rate = 0.1
crossover_rate = 0.8

best_fitness = genetic_algorithm(population_size, lower_bound, upper_bound, generations,
mutation_rate, crossover_rate)
print(f"Best fitness found: {best_fitness:.4f}")
```

OUTPUT:

```
Generation 1: Best Fitness = 84.7106
Generation 2: Best Fitness = 85.5059
Generation 3: Best Fitness = 85.3008
Generation 4: Best Fitness = 93.7781
Generation 5: Best Fitness = 100.0000
Generation 6: Best Fitness = 97.6004
Generation 7: Best Fitness = 97.6004
Generation 8: Best Fitness = 94.8915
Generation 9: Best Fitness = 84.8943
Best fitness found: 81.8096
```

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:

25/11/24

LAB-04:

Particle swarm optimization

Algorithm:

Step1: Pick a mathematical fn $f(x)$ to optimize

Step2: Set parameters N, ω, c1, c2 where c1 is weight of personal best position and c2 is weight of global best position.

Step3: Define the limits within which particle can move

Step4: Assign N with random velocity

Step5: for each particle calculate its fitness that is the best position

Step6: Update velocity based on the best velocity of its own and based on the best velocity found by the swarm.

Step7: It undergoes iterations to check the best solution found

Step8: Then in the final iteration it finds out the best value

Pseudocode:

velocity = w * velocity + c1 * r1 * (best_position - position) + c2 * r2 * (global_best_position - position)

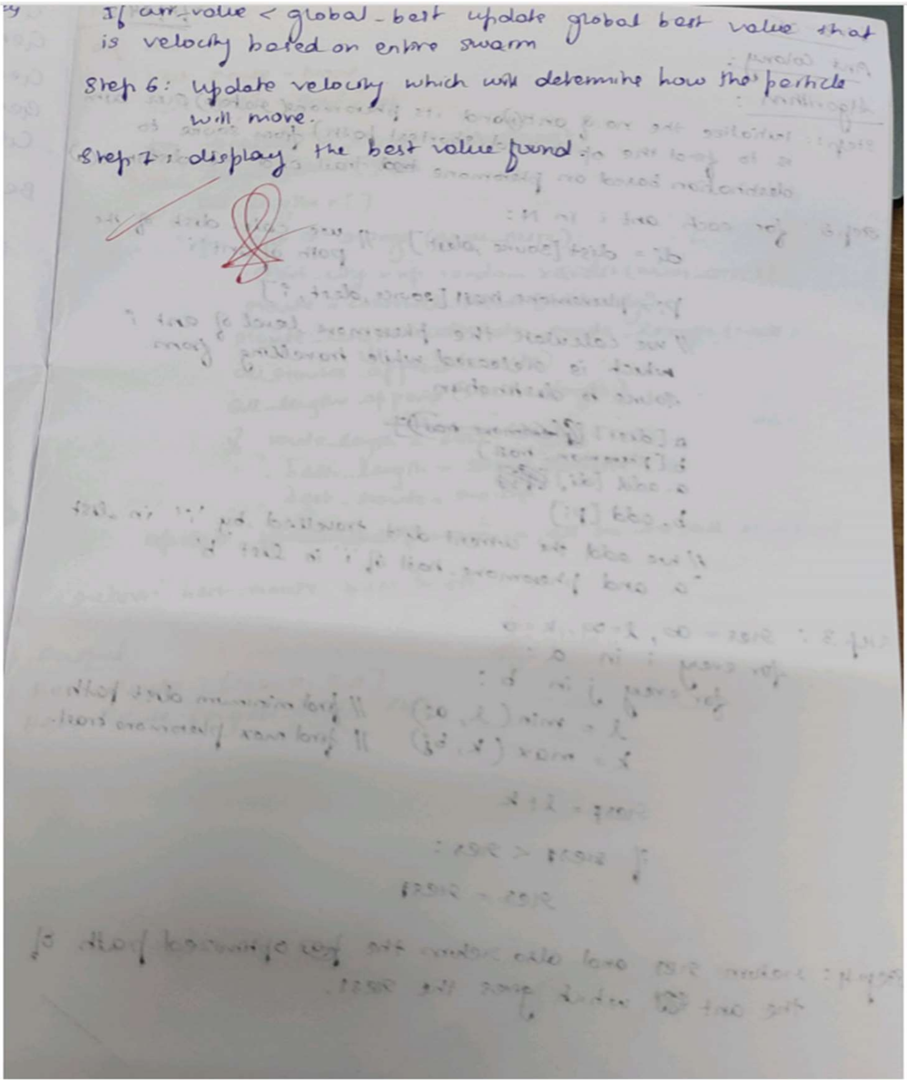
step1: def func(x)
return x**2

step2: Initialize parameters
N = 30, W = 0.5, c1 = 1.5, c2 = 1.5

step3: for each particle initialize them position and velocity randomly within the range [-10, 10], [-1, 1]

step4: In this step we evaluate the fitness by sending values to function

step5: Updating values of curr_value < best_value
update the best of its own particle


If curr_value < global_best update global best value that is velocity based on entire swarm

Step 6: Update velocity which will determine how the particle will move.

Step 7: display the best value found.

Code:

```
import numpy as np

def objective_function(x):
    return x**2

def particle_swarm_optimization(obj_func, num_particles=30, num_iterations=5, bounds=(-10, 10), w=0.5, c1=1.5, c2=1.5):
```

```
    positions = np.random.uniform(bounds[0], bounds[1], num_particles)
    velocities = np.random.uniform(-1, 1, num_particles)
    personal_best_positions = positions.copy()
    personal_best_scores = obj_func(personal_best_positions)
    global_best_position = personal_best_positions[np.argmin(personal_best_scores)]
    global_best_score = obj_func(global_best_position)

    for iteration in range(num_iterations):
        for i in range(num_particles):
            r1, r2 = np.random.rand(), np.random.rand()
            velocities[i] = (
                w * velocities[i]
                + c1 * r1 * (personal_best_positions[i] - positions[i])
                + c2 * r2 * (global_best_position - positions[i])
            )
            positions[i] += velocities[i]
            positions[i] = np.clip(positions[i], bounds[0], bounds[1])
            score = obj_func(positions[i])
            if score < personal_best_scores[i]:
                personal_best_positions[i] = positions[i]
                personal_best_scores[i] = score

        best_particle_index = np.argmin(personal_best_scores)
        if personal_best_scores[best_particle_index] < global_best_score:
            global_best_position = personal_best_positions[best_particle_index]
            global_best_score = personal_best_scores[best_particle_index]

        print(f"Iteration {iteration+1}/{num_iterations}, Global Best Score: {global_best_score}")

    return global_best_position, global_best_score

best_position, best_score = particle_swarm_optimization(objective_function)
```

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

OUTPUT:

```
Iteration 1/5, Global Best Score: 0.0035074795348310614
Iteration 2/5, Global Best Score: 0.00245103757583387
Iteration 3/5, Global Best Score: 0.0005017063107877815
Iteration 4/5, Global Best Score: 0.0005017063107877815
Iteration 5/5, Global Best Score: 0.0005017063107877815

Optimization Results:
Best Position: 0.022398801548024427
Best Score: 0.0005017063107877815
```

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:

LAB-05 8/11/24

Ant Colony:

Algorithm:

step 1: Initialize the no. of ants(N) and its pheromone rate(P). Our aim is to find the optimized (shortest path) from source to destination based on pheromone ~~trail~~ trail and the distance(D)

step 2: for each ant i in N:

di = dist [source, dest] // we calc. dist of the path of ant 'i'

pi = pheromone_trail [source, dest, i]

// we calculate the pheromone level of ant i which is released while travelling from source to destination.

a [dist] ~~[pheromone_trail]~~
b [pheromon_trail]
a. add [di] ~~[pi]~~
b. add [pi]

// we add the current dist. travelled by 'i' in list 'a' and pheromone_trail of 'i' in list 'b'.

step 3: res = ∞, l = ∞, k = 0

for every i in a:
  for every j in b:
    l = min (l, ai) // find minimum dist. path
    k = max (k, bj) // find max pheromone trail.

  res1 = l + k

  if res1 < res:
    res = res1

step 4: return res and also return the ~~[illegible]~~ optimized path of the ant ~~[illegible]~~ which gives the res1.

Pseudocode:

```
def ACO():
    best_route = None
    best_length = float('inf')

    for iteration in range(ITERATIONS):
        all_routes = []
        all_lengths = []

        for ant in range(NUM_ANTS):
            start_city = np.random.randint(NUM_CITIES)
            route = construct_route(start_city)
            route_length = calculate_route_length(route)
            all_routes.append(route)
            all_lengths.append(route_length)
            if route_length < best_length:
                best_length = route_length
                best_route = route

    update_pheromone(pheromone, all_routes, all_lengths)

    return best_route, best_length.
```

8/11/24

Output:

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

Best Route Length : 227.345

Code:

```
import numpy as np

NUM_CITIES = 10
NUM_ANTS = 20
ITERATIONS = 10
ALPHA = 1.0
BETA = 2.0
EVAPORATION_RATE = 0.5
Q = 100

distance_matrix = np.random.randint(1, 100, size=(NUM_CITIES, NUM_CITIES))
np.fill_diagonal(distance_matrix, 0)
pheromones = np.ones((NUM_CITIES, NUM_CITIES))

def calculate_route_length(route):
    length = 0
    for i in range(len(route) - 1):
        length += distance_matrix[route[i], route[i + 1]]
    length += distance_matrix[route[-1], route[0]]
    return length

def construct_route(start_city):
    route = [start_city]
    for _ in range(NUM_CITIES - 1):
        current_city = route[-1]
        probabilities = []
        for next_city in range(NUM_CITIES):
            if next_city not in route:
                prob = (pheromones[current_city, next_city] ** ALPHA) * \
```

```
                ((1 / distance_matrix[current_city, next_city]) ** BETA)
            probabilities.append(prob)
        else:
            probabilities.append(0)
    probabilities = np.array(probabilities)
    probabilities /= probabilities.sum()
    next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
    route.append(next_city)
return route

def update_pheromones(pheromones, all_routes, all_lengths):
    pheromones *= (1 - EVAPORATION_RATE)
    for route, length in zip(all_routes, all_lengths):
        pheromone_deposit = Q / length
        for i in range(len(route) - 1):
            pheromones[route[i], route[i + 1]] += pheromone_deposit
            pheromones[route[i + 1], route[i]] += pheromone_deposit
        pheromones[route[-1], route[0]] += pheromone_deposit
        pheromones[route[0], route[-1]] += pheromone_deposit

def aco():
    best_route = None
    best_length = float('inf')
    for _ in range(ITERATIONS):
        all_routes = []
        all_lengths = []
        for _ in range(NUM_ANTS):
            start_city = np.random.randint(0, NUM_CITIES)
            route = construct_route(start_city)
            route_length = calculate_route_length(route)
            all_routes.append(route)
            all_lengths.append(route_length)
```

```
            if route_length < best_length:
                best_length = route_length
                best_route = route
        update_pheromones(pheromones, all_routes, all_lengths)
    return best_route, best_length

best_route, best_length = aco()
print("Best Route:", best_route)
print("Best Length:", best_length)
```

OUTPUT:

```
Best Route: [1, np.int64(4), np.int64(7), np.int64(2), np.int64(3), np.int64(5), np.int64(8),
 np.int64(9), np.int64(6), np.int64(0)]
Best Length: 226
```

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:

15/4/24

LAB - 06:

Cuckoo Search:

Algorithm:

Step 1: Define the parameters
n = no. of nests / solutions
i = no. of iterations
Pa = probability of finding cukoo's egg in that particular nest
Also define the range of the ley flight (upper & lower bound)

Step 2: Randomly assign each nest with some eggs.

Step 3: best_nest = nest[0]
~~prob~~ prob = prob[0]

Step 4: while (iterations != 0 && flight is within ley range)
{
// calculate the probability of finding the egg in the current nest
curr = prob[i]
nest = nest[i]
// calculating best nest
if ~~nest~~ < best_nest:
best_nest = nest
~~prob = curr~~ prob = curr

iterations --

}

Step 5: print("The best nest :" : ~~nest~~ best_nest)
print("The probability of finding an egg there" : prob)

Study algorithm implementation in wireless N/w

→ Define f(x) which we need to optimize
→ this algo. allows us to explore large areas of search space
↳ promotes diversity
→ it evaluates the fitness of each nest, maximization problem
↳ higher fitness corresponds to better solution

• egg laying
• Levy flight
• Replace eggs
• Fitness

Wireless Networks:
Used in WSN → aim is to manage limited resources ~~(battery power)~~ (bandwidth) efficiently. network stability

→ Load balancing
problem: even distribution
Soln: exploring various config.

→ Resource Allocation
problem: bandwidth
Soln: max throughput while min congestion

Output:
Generation 1/3 – Best Fitness: 406.17
Generation 2/3 – Best Fitness: 406.17
Generation 3/3 – Best Fitness: 406.17
optimized load distribution: [57, 30.1, 26.0, 57, 6.4, 30]

8/10
15/11

Code:

```
import numpy as np

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

def levy_flight(Lambda):
    u = np.random.normal(0, 1, 1)
    v = np.random.normal(0, 1, 1)
    step = u / abs(v)**(1 / Lambda)
    return step

def cuckoo_search(n, max_iter, dim, bounds, pa=0.25, alpha=0.01, Lambda=1.5):
    nests = np.random.uniform(bounds[0], bounds[1], (n, dim))
    fitness = np.array([objective_function(nest) for nest in nests])
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(n):
            step_size = alpha * levy_flight(Lambda)
            new_nests[i] += step_size * (nests[i] - best_nest)
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

        new_fitness = np.array([objective_function(nest) for nest in new_nests])
        for i in range(n):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]
```

```
        for i in range(n):
            if np.random.rand() < pa:
                nests[i] = np.random.uniform(bounds[0], bounds[1], dim)
                fitness[i] = objective_function(nests[i])

        best_nest = nests[np.argmin(fitness)]
        best_fitness = np.min(fitness)

        print(f"Iteration {iteration + 1}/{max_iter}, Best Fitness: {best_fitness:.4f}")

    return best_nest, best_fitness

n = 20
max_iter = 10
dim = 5
bounds = [-10, 10]
pa = 0.25
alpha = 0.01

best_solution, best_fitness = cuckoo_search(n, max_iter, dim, bounds, pa, alpha)

print(f"Best Solution: {best_solution}")
print(f"Best Fitness (Optimized Energy): {best_fitness:.4f}")
```

OUTPUT:

```
Iteration 1/10, Best Fitness: 49.6197
Iteration 2/10, Best Fitness: 49.6197
Iteration 3/10, Best Fitness: 69.3300
Iteration 4/10, Best Fitness: 49.4658
Iteration 5/10, Best Fitness: 49.4658
Iteration 6/10, Best Fitness: 34.6011
Iteration 7/10, Best Fitness: 34.6011
Iteration 8/10, Best Fitness: 61.2966
Iteration 9/10, Best Fitness: 61.2966
Iteration 10/10, Best Fitness: 49.4222
Best Solution: [ 3.77706023  1.36717565 -5.02020857 -1.99481751 -2.02609847]
Best Fitness (Optimized Energy): 49.4222
```

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:

22/11/24

LAB-07:

Grey Wolf Optimizer

N = no. of wolves
T = total no. of iterations
D = no. of dimensions of search space

alpha, beta, delta = none

Evaluate the fitness of each wolf and assign the best three to alpha, beta and delta wolves.

a = 2 // Initialize the coefficient

for t=1 to T :
    for each wolf i in wolves :
        for each d in dimensions D :
            $A_1 = 2 * a * random() - a$
            $B_1 = 2 * a$
            $X_1 = abs(2 * wolves[i,d] * B_1 - A_1)$

            $A_2 = 2 * a * random() - a$
            $B_2 = 2 * a$
            $X_2 = abs(2 * wolves[i,d] * B_2 - A_2)$

            $A_3 = 2 * a * random() - a$
            $B_3 = 2 * a$
            $X_3 = abs(2 * wolves[i,d] * B_3 - A_3)$

            $X_i[d] = (X_1 + X_2 + X_3) / 3$ //Stores the average values for each i

        (end for)
    (end for)
    Evaluate the best wolves (alpha, beta, delta) for each iteration.
    $a = 2(1 - (t/T))$
(end for)

Return the alpha wolf as the best solution.

Implementation
WOT - Image Processing
Disadv

10/10

24

Disadvantages:

→ Scalability issues → struggles with high dimensional problems

→ High Sensitivity to parameters → pop size, iteration count.

→ Premature convergence → converges too quickly to local optimum especially in complex problems.

→ No Guarantee of optimality → doesnot not guarantee finding global optimum

To overcome these disadvantages, we can use PSO-GWO

→ PSO's velocity & position update → to maintain diversity & explore

→ GWO's heirarchical mech. for exploitation to refine solutions.

Code:

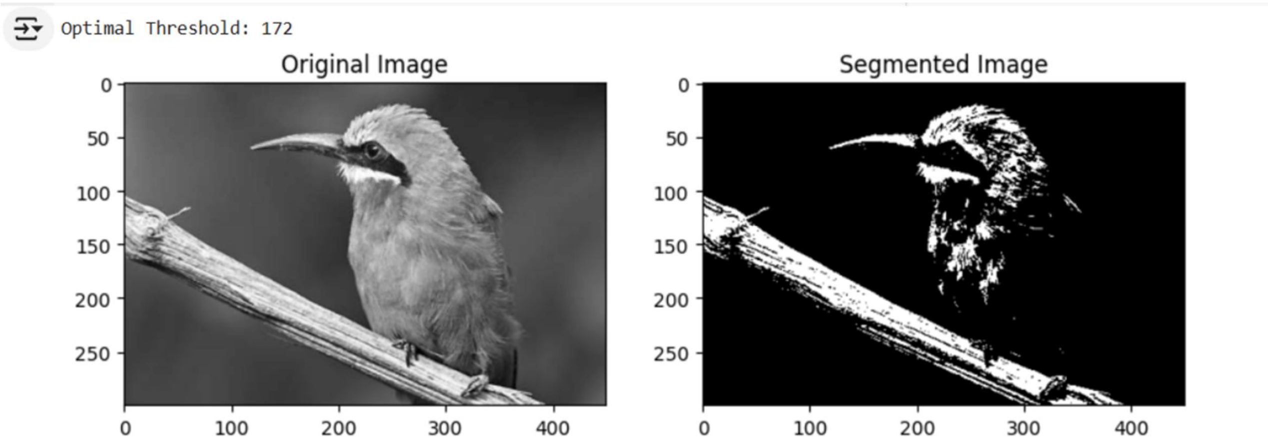
```
import numpy as np
import cv2
import matplotlib.pyplot as plt
```

```
def otsu_variance(threshold, histogram, total_pixels):
    background_weight = np.sum(histogram[:threshold])
    foreground_weight = np.sum(histogram[threshold:])
    if background_weight == 0 or foreground_weight == 0:
        return float('inf')
    background_mean = np.sum(np.arange(threshold) * histogram[:threshold]) / background_weight
    foreground_mean = np.sum(np.arange(threshold, 256) * histogram[threshold:]) / foreground_weight
    between_class_variance = background_weight * foreground_weight * (background_mean - foreground_mean) ** 2
    return -between_class_variance

def grey_wolf_optimizer(histogram, total_pixels, max_iter=50, population_size=10):
    dim = 1
    alpha_pos, beta_pos, delta_pos = None, None, None
    alpha_score, beta_score, delta_score = float('inf'), float('inf'), float('inf')
    wolves = np.random.randint(0, 256, (population_size, dim))

    a = 2
    for iteration in range(max_iter):
        for i in range(population_size):
            fitness = otsu_variance(wolves[i][0], histogram, total_pixels)
            if fitness < alpha_score:
                alpha_score, beta_score, delta_score = fitness, alpha_score, beta_score
                alpha_pos, beta_pos, delta_pos = wolves[i], alpha_pos, beta_pos
            elif fitness < beta_score:
                beta_score, delta_score = fitness, beta_score
                beta_pos, delta_pos = wolves[i], beta_pos
            elif fitness < delta_score:
                delta_score = fitness
                delta_pos = wolves[i]

        for i in range(population_size):
```

```
            for d in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()
                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * alpha_pos[d] - wolves[i][d])
                X1 = alpha_pos[d] - A1 * D_alpha

                r1, r2 = np.random.rand(), np.random.rand()
                A2, C2 = 2 * a * r1 - a, 2 * r2
                D_beta = abs(C2 * beta_pos[d] - wolves[i][d])
                X2 = beta_pos[d] - A2 * D_beta

                r1, r2 = np.random.rand(), np.random.rand()
                A3, C3 = 2 * a * r1 - a, 2 * r2
                D_delta = abs(C3 * delta_pos[d] - wolves[i][d])
                X3 = delta_pos[d] - A3 * D_delta

                wolves[i][d] = np.clip((X1 + X2 + X3) / 3, 0, 255)

        a -= 2 / max_iter

    return int(alpha_pos[0])

if __name__ == "__main__":
    img = cv2.imread("/content/design_resolution_original.jpg", 0)
    histogram, _ = np.histogram(img.ravel(), bins=256, range=(0, 256))
    total_pixels = img.size

    optimal_threshold = grey_wolf_optimizer(histogram, total_pixels)
    print("Optimal Threshold:", optimal_threshold)

    _, segmented_img = cv2.threshold(img, optimal_threshold, 255, cv2.THRESH_BINARY)
```

```
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.title("Original Image")
plt.imshow(img, cmap="gray")
plt.subplot(1, 2, 2)
plt.title("Segmented Image")
plt.imshow(segmented_img, cmap="gray")
plt.show()
```

OUTPUT:



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:

LAB-08:

Parallel Cellular Algorithms

Algorithm

```
Function obj_fun(x):
    return x**2 - 4*x + 4
Function initialize_pop():
    create random grid with values in range [-10, 10], return grid.
Function evaluate_fitness(population):
    create empty array fitness
    for each cell in popl^n:
        fitness[cell] = obj_fun(pop[cell])
    return fitness
Function update_cell(pop, fitness, neigh_states):
        temp ← pop, neigh_radius neigh_state
neighbourhood = []
  for di from (-neigh_rad to neigh_rad)
     for dj from " :
          ni = i+di , nj = j+dj
          if (ni, nj) ≤ bounds:
               add (pop^n [ni, nj])
  sort neighbourhood
  best_neigh = neighbour[0]
algorithm():
   pop ← initialize_pop()
   fitness ← eval_fitness(pop)
   best_sol ← None
   best_fit ← infinity
for each iteration from i to iterations:
      new pop ← update_cell()
      fitness ← evaluate()
      find min(fitness)
      if min_fitness < best_fit:
           best_fit = min_fitness
           best_sol ← sol^n in new popl^n
```

Implementation:

Image Processing:

→ Edge Detection:

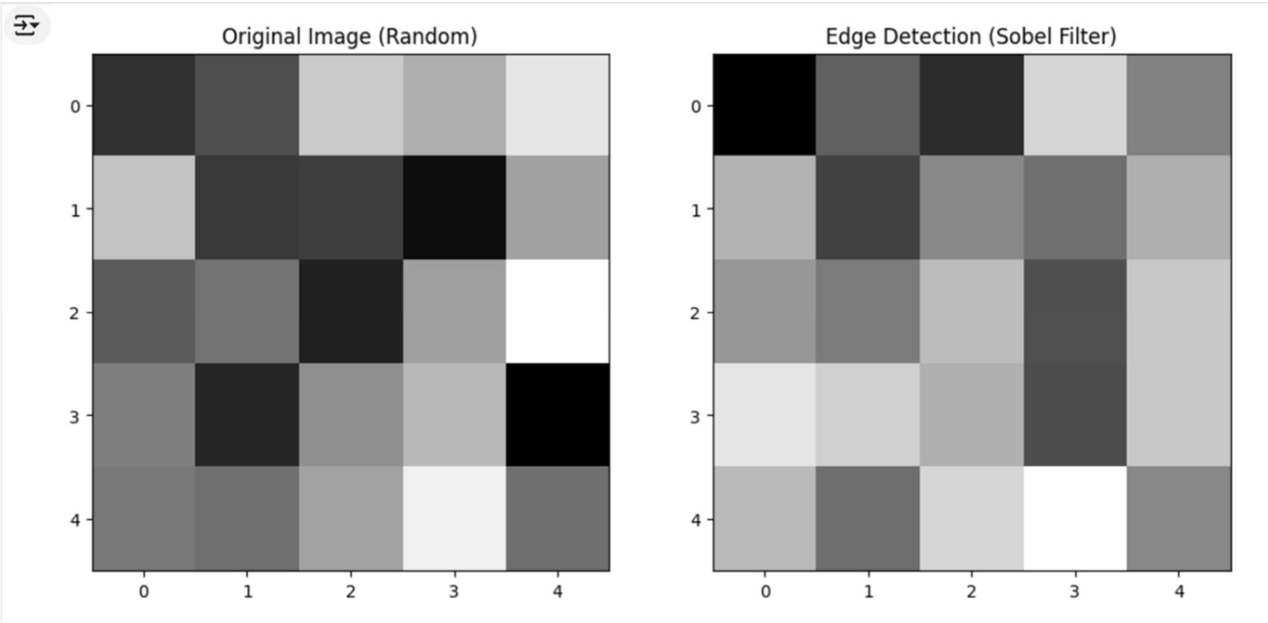
Identifies significant changes in pixel intensity that correspond to the boundaries of objects or features in an image.

Code:

```
import numpy as np
from scipy.ndimage import convolve
import matplotlib.pyplot as plt
```

```
rows, cols = 5, 5
grid = np.random.randint(0, 255, size=(rows, cols), dtype=np.uint8)

sobel_x = np.array([[-1, 0, 1],
                    [-2, 0, 2],
                    [-1, 0, 1]])

sobel_y = np.array([[-1, -2, -1],
                    [ 0,  0,  0],
                    [ 1,  2,  1]])

def apply_filter(grid, kernel):
    return convolve(grid, kernel, mode='constant', cval=0)

def update_grid(grid):
    edges_x = apply_filter(grid, sobel_x)
    edges_y = apply_filter(grid, sobel_y)
    new_grid = np.hypot(edges_x, edges_y)
    new_grid = (new_grid / new_grid.max()) * 255
    return new_grid.astype(np.uint8)

plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.title("Original Image (Random)")
plt.imshow(grid, cmap='gray')

new_grid = update_grid(grid)

plt.subplot(1, 2, 2)
plt.title("Edge Detection (Sobel Filter)")
plt.imshow(new_grid, cmap='gray')
plt.show()
```

OUTPUT:



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:

Optimization via Gene Expression Algorithms:

Algorithm:

```
Function obj_fun(x, y)
    return x**2 + y**2
Function initialize_pop():
    for i in size:
        pop[i] ← random value

Function evaluate_fitness(pop):
    fitness ← []
    for ind in pop:
        fitness[ind] = obj_fn()
    return fitness
Function select_parents(pop, fitness):
    prob ← 1/fitness - e
    prob ← sum(prob)
    indices ← random set of indices from population
    return pop[indices]
Function crossover(par1, par2):
    if random < crossover_rate:
        point ← random from 1 to num
        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_:
        if random < mut_rate:
            ind[i] = ind[i] + random(-1,1)
    return individual
```

Implementation:

Output:

Generation 1 : Best Fitness = 1.9219
Generation 2 : Best Fitness = 3.8054
Generation 3 : Best Fitness = 0.4556
Generation 4 : Best Fitness = 0.6357
Generation 5 : Best Fitness = 0.9917

Differences b/n Genetic Algo & Gene Exp Algo:

Genetic Algo:

→ sol^ns are rep as strings / binary dgts
→ swapping parts of 2 chromosomes
→ works well for optimizing fixed structures & req. more effort to maintain diversity

Gene Exp Algo:

→ rep as expression trees or linear chromosomes
→ exchanging sub trees between chromosomes
→ more suited for problems where the sol^ns are not fixed & performs better compared to genetic algo.

Code:

```
import random

POPULATION_SIZE = 100
GENERATIONS = 5
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.7
MAX_TREE_DEPTH = 5
FUNCTIONS = ['+', '*', '/']
TERMINALS = ['x', '1', '2', '3']

class Individual:
    def __init__(self, expression):
        self.expression = expression
        self.fitness = float('inf')

    def evaluate_fitness(self, x_value):
        try:
            expr = self.expression.replace('x', str(x_value))
            self.fitness = eval(expr)
        except Exception:
            self.fitness = float('inf')

def generate_random_individual():
    expression = generate_random_expression(MAX_TREE_DEPTH)
    return Individual(expression)

def generate_random_expression(depth):
    if depth == 0 or random.random() < 0.3:
```

```
        return random.choice(TERMINALS)
    else:
        function = random.choice(FUNCTIONS)
        left = generate_random_expression(depth - 1)
        right = generate_random_expression(depth - 1)
        return f"({left} {function} {right})"

def crossover(parent1, parent2):
    expr1, expr2 = parent1.expression, parent2.expression
    split1 = random.choice(expr1.split())
    split2 = random.choice(expr2.split())
    offspring_expr = expr1.replace(split1, split2, 1)
    return Individual(offspring_expr)

def mutate(individual):
    if random.random() < MUTATION_RATE:
        mutated_expr = individual.expression
        split_expr = mutated_expr.split()
        mutated_expr = mutated_expr.replace(random.choice(split_expr),
generate_random_expression(MAX_TREE_DEPTH), 1)
        individual.expression = mutated_expr

def select_best_individual(population, x_value):
    best_individual = min(population, key=lambda ind: ind.fitness)
    best_individual.evaluate_fitness(x_value)
    return best_individual

def run_gep_algorithm():
    population = [generate_random_individual() for _ in range(POPULATION_SIZE)]

    for generation in range(GENERATIONS):
        for individual in population:
```

```
            individual.evaluate_fitness(3)

        best_individual = select_best_individual(population, 3)
        print(f"Generation {generation + 1}: Best fitness = {best_individual.fitness}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            if random.random() < CROSSOVER_RATE:
                parent1 = random.choice(population)
                parent2 = random.choice(population)
                offspring = crossover(parent1, parent2)
                new_population.append(offspring)
            else:
                individual = random.choice(population)
                mutate(individual)
                new_population.append(individual)

        population = new_population

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

OUTPUT:

```
Generation 1: Best fitness = 0.004208754208754209
Generation 2: Best fitness = 0.006687242798353909
Generation 3: Best fitness = 0.00041305245766212 31
Generation 4: Best fitness = 8.742022904100009e-05
Generation 5: Best fitness = 1
```