VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PRAVEEN RATHOD (1BM23CS248)

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
Aug-2025 to Jan-2026

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 **PRAVEEN RATHOD (1BM23CS248),** 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. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/Praveen4584/1BM23CS248-BIS-LAB

Observation Index:

INDEX

Name : Praveen Rathod Class : 3rd year

Section : 5F Roll No. : 248 Subject : Bio Inspired System.

Sl. No.	Date	Title	Page No.	Teacher's Sign. / Remarks
1	21-8-25	Description of algorithms.	1-6	—
2	28-8-25	genetic algorithm for optimi -zation problems	7-9	10
3	4/9/25	optimization via gene Expression Algorithms.	10-12	10
4	11/9/25	Particle Swarm optimization	13-15	9
5	9/10/25	Ant colony optimization (ACO) for the travelling salesman pblm.	16-17	10
6	16/10/25	cuckoo search (CS)	18-19	9
7	23/10/25	grey wolf optimizer (GWO)	20-21	10
8.	30/10/25	parallel cellular algorithm & program.	22-23	10

Program 1

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.

Implementation Steps:

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

2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.

3. Create Initial Population: Generate an initial population of potential solutions.

4. Evaluate Fitness: Evaluate the fitness of each individual in the population.

5. Selection: Select individuals based on their fitness to reproduce.

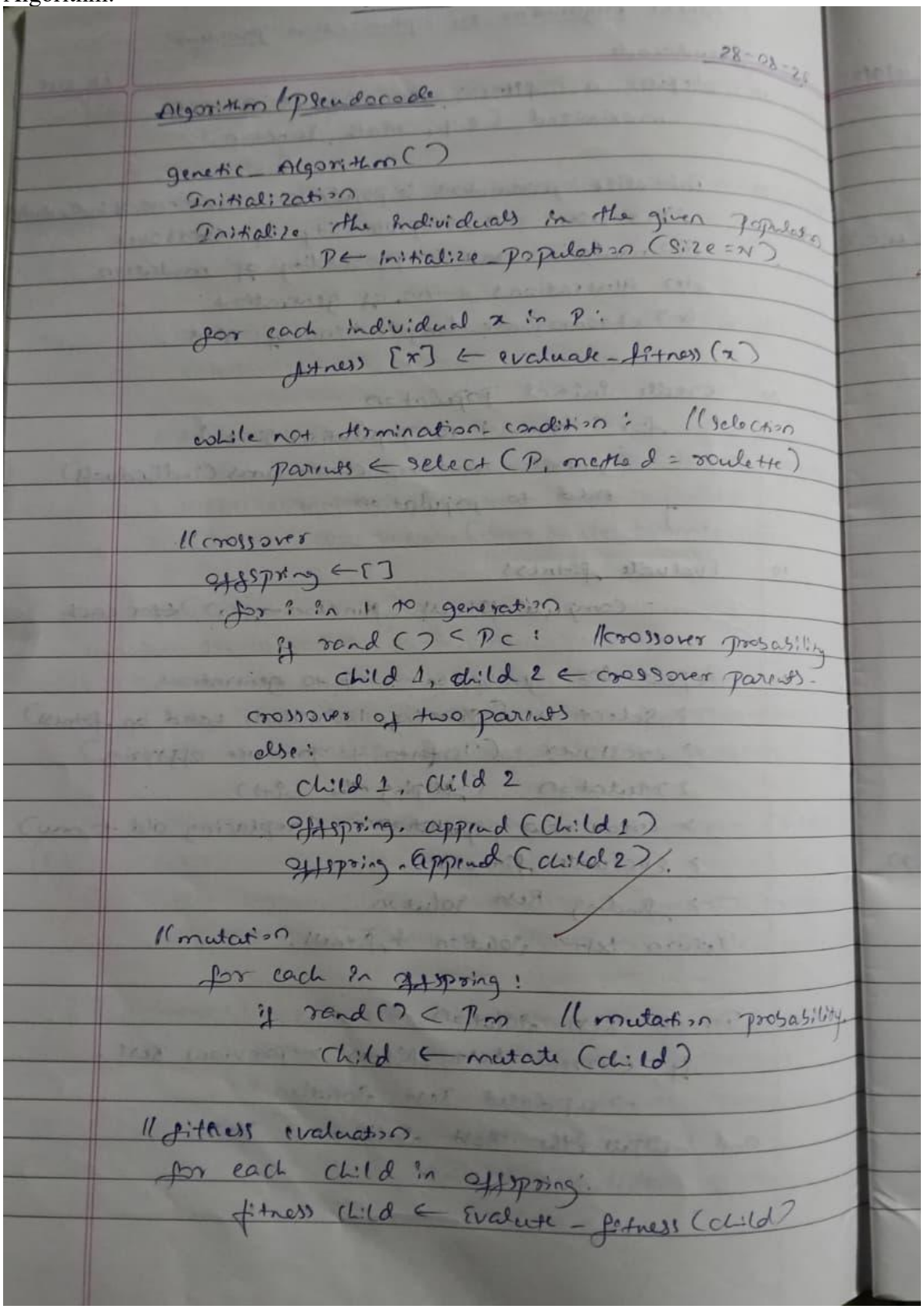
6. Crossover: Perform crossover between selected individuals to produce offspring.

7. Mutation: Apply mutation to the offspring to maintain genetic diversity.

8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.

9. Output the Best Solution: Track and output the best solution found during the generations.

Algorithm:



28-08-25

Algorithm / Pseudocode

```
Genetic_Algorithm()
Initialization
Initialize the individuals in the given population
    P ← initialize_population(size=N)

for each individual x in P:
    fitness[x] ← evaluate_fitness(x)

while not termination_condition:  // selection
    parents ← select(P, method = roulette)

// crossover
offspring ← []
for i in 1 to generation
    if rand() < Pc:  // crossover probability
        child 1, child 2 ← crossover parents.
crossover of two parents
    else:
        child 1, child 2
    offspring.append(child 1)
    offspring.append(child 2).

// mutation
for each in offspring:
    if rand() < Pm  // mutation probability
        child ← mutate(child)

// fitness evaluation
for each child in offspring
    fitness child ← Evaluate_fitness(child)
```

// termination

P ← selection of survivor (P, offspring)

best ← fitness [x], over x ∈ P

return best.

- Events in the field of image Processing.

Output:-

Gen 0: Best fitness = 9327346.1667,
Params = [1.63810974 0.57168993 1.75007855]

Gen 1: Best fitness = 9424575.7286,
Params = [1.36973675 0.5 2.]

Gen 2: Best fitness = 9635870.0394,
Params = [1.97500278 0.52827145 1.49107197]

Gen 3: Best fitness = 9760056.3209
Params = [1.82492376 0.5 1.51500324]

Gen 4: Best fitness = 9798367.6403
Params = [2. 0.5 1.73567691]

Gen 5: Best fitness = 9798367.6403
Params = [2. 0.5. 1.73567691]

Gen 6: Best fitness = 9798367.6403
Params = [2. 0.5. 1.73567691]

Best parameters found: [2. 0.5 1.73567691]

Code:

```
import random
import numpy as np
from PIL import Image, ImageEnhance
from skimage.metrics import structural_similarity as ssim
from skimage.filters import sobel
from skimage import img_as_float
# ---------- Parameters ----------
POP_SIZE = 20
N_GEN = 15
MUT_RATE = 0.3
ELITE = 2  # keep best individuals each gen

# Load image
original = Image.open("input.jpg").convert("RGB")
original_np = np.array(original)

# If you have a reference image, load it
try:
    target = Image.open("target.jpg").convert("RGB")
    target_np = np.array(target)
    USE_REFERENCE = True
except:
    USE_REFERENCE = False

# ---------- Apply Adjustments ----------
def apply_adjustments(params):
    b, c, s = params
    img = original.copy()
    img = ImageEnhance.Brightness(img).enhance(b)
    img = ImageEnhance.Contrast(img).enhance(c)
    img = ImageEnhance.Sharpness(img).enhance(s)
    return img

# ---------- Fitness Function ----------
def fitness(params):
    img = apply_adjustments(params)
    img_np = np.array(img)

    if USE_REFERENCE:
        return ssim(target_np, img_np, channel_axis=2)
    else:
        gray = img.convert("L")
        gray_np = img_as_float(np.array(gray))
        entropy = -np.sum(gray_np * np.log2(gray_np + 1e-10))
        edge_strength = np.mean(sobel(gray_np))
        return entropy + edge_strength
```

```
# ---------- GA Core ----------
def init_population(size):
    return [np.array([
        random.uniform(0.5, 2.0),  # Brightness
        random.uniform(0.5, 2.0),  # Contrast
        random.uniform(0.5, 2.0)   # Sharpness
    ]) for _ in range(size)]

def crossover(p1, p2):
    alpha = random.random()
    return alpha * p1 + (1 - alpha) * p2

def mutate(ind):
    if random.random() < MUT_RATE:
        ind += np.random.normal(0, 0.2, size=3)
        ind = np.clip(ind, 0.5, 2.0)
    return ind

def select(pop, fitnesses):
    i, j = random.sample(range(len(pop)), 2)
    return pop[i] if fitnesses[i] > fitnesses[j] else pop[j]

# ---------- Run GA ----------
def run_ga():
    pop = init_population(POP_SIZE)

    for gen in range(N_GEN):
        fitnesses = [fitness(ind) for ind in pop]
        ranked = sorted(zip(pop, fitnesses), key=lambda x: x[1], reverse=True)
        best_ind, best_fit = ranked[0]

        print(f"Gen {gen}: Best fitness = {best_fit:.4f}, Params = {best_ind}")

        new_pop = [ind.copy() for ind, _ in ranked[:ELITE]]

        while len(new_pop) < POP_SIZE:
            p1, p2 = select(pop, fitnesses), select(pop, fitnesses)
            child = crossover(p1, p2)
            child = mutate(child)
            new_pop.append(child)

        pop = new_pop

    best_img = apply_adjustments(best_ind)
    best_img.save("optimized.jpg")
    print("Best parameters found:", best_ind)
    display(best_img)  # Show inside Jupyter

run_ga()
```

Program 2

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.

Implementation Steps:

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

2. Initialize Parameters: Set the population size, number of genes, mutation rate, crossover rate, and number of generations.

3. Initialize Population: Generate an initial population of random genetic sequences.

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

5. Selection: Select genetic sequences based on their fitness for reproduction.

6. Crossover: Perform crossover between selected sequences to produce offspring.

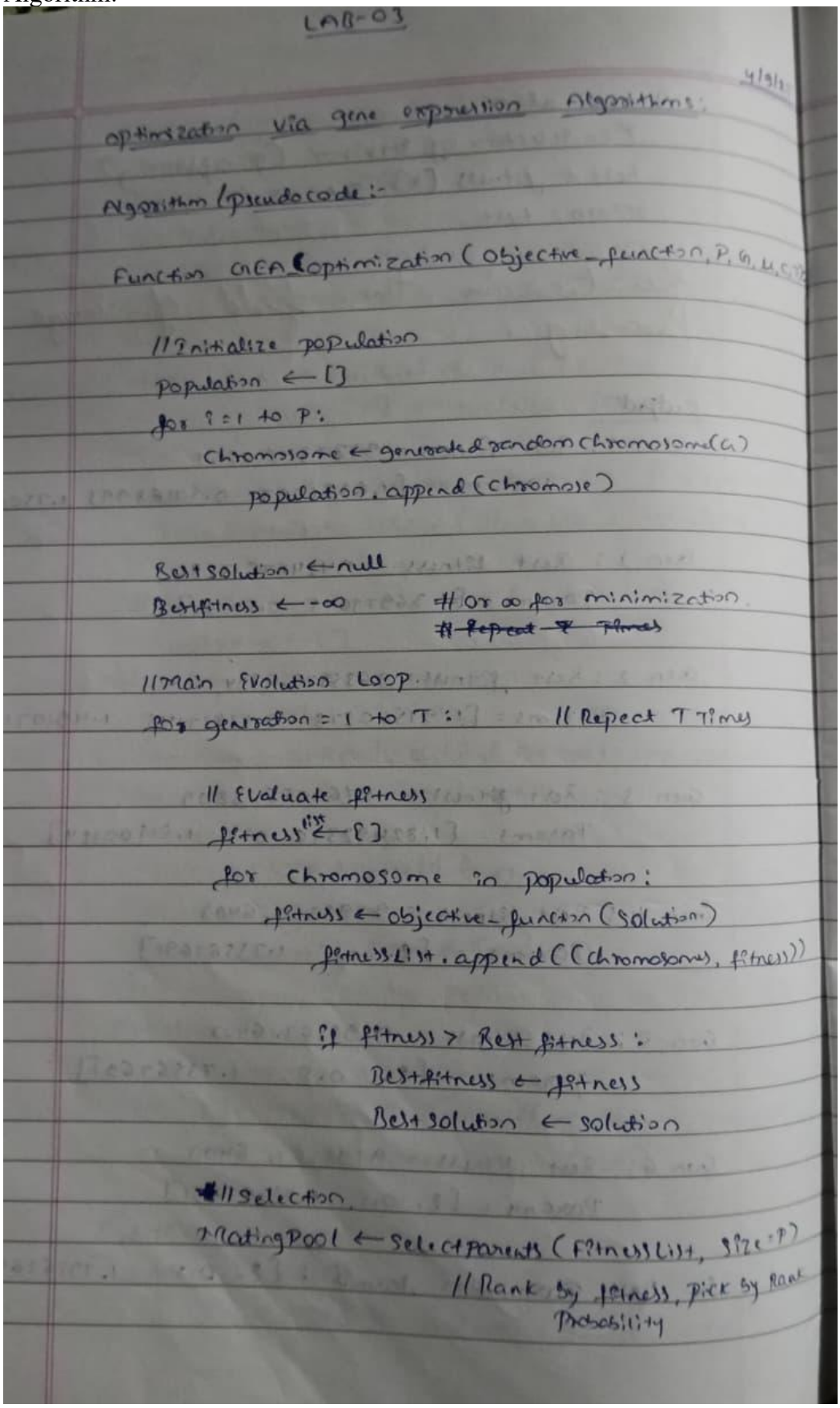
7. Mutation: Apply mutation to the offspring to introduce variability.

8. Gene Expression: Translate genetic sequences into functional solutions.

9. Iterate: Repeat the selection, crossover, mutation, and gene expression processes for a fixed number of generations or until convergence criteria are met.

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

Algorithm:



LAB-03

4/9/25

optimization via gene expression Algorithms:

Algorithm / pseudocode :-

Function GEA_optimization (objective_function, P, G, U, C, T)

//Initialize population
Population ← []
for i : 1 to P :
    chromosome ← generate_random_chromosome(G)
    population.append(chromosome)

Best_solution ← null
Best_fitness ← -∞ # or ∞ for minimization
Repeat T times

//Main Evolution Loop
for generation : 1 to T : // Repeat T times

    // Evaluate fitness
    fitness_list ← []
    for chromosome in population :
        fitness ← objective_function (solution)
        fitness_list.append ((chromosome, fitness))

        if fitness > Best_fitness :
            Best_fitness ← fitness
            Best_solution ← solution

    // Selection
    MatingPool ← SelectParents (FitnessList, Size: P)
                // Rank by fitness, pick by Roat
                Probability

// crossover & Mutation → offspring
offSpring ← [ ]
for i = 0 to P // main evolution loop :
    parent 1 ← Mating pool [i]
    parent 2 ← Mating pool [i+1]

    if Random ( ) < C :
        child 1 , child 2 ← copy (parent 1),
                            copy (parent 2)

        Mutate (child 1, μ)
        Mutate (child 2, μ)

        offSpring. append (child 1)
        offspring. append (child 2)

// Replace old population
population ← offspring

// Return best result
Return Best Solution , Best fitness .

Note:- Objective function :- a function that takes a solution and returns a scalar score.

P ← population size [20-500]
G ← genome size [no. of G / chromosomes]
μ ← mutation rate
C ← crossover probability
T ← generations [50-500+]

Cond's / GE → "Fails", any specific fel field.

output:

Gen 1 : Best fitness = -80, Best solution = [3,2,0,1] [0,1,3,2]
Gen 11 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 21 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 31 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 41 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 51 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 61 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 71 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 81 : Best fitness = -80, Best solution = [3,2,0,1]
Gen 91 : Best fitness = -80, Best solution = [3,2,0,1]

Best solution found : [3,2,0,1]
Best fitness (Negative Tour length) : -80.

Why it might fail :->

1) Chromosome representation is designed specifically for TSP, which is a permutation problem. If you change the problem (like Knapsack), the chromosome structure and genetic operators would need to be adjusted.

-> If you apply the specific TSP-code to any other problem (like Knapsack / Job scheduling, etc). It will FAIL unless you adjust the chromosome encoding, objective function and genetic operators to match the new problem.

Code:

```
import random
import numpy as np

# -------------------- Objective Function (TSP) --------------------
def objective_function(tour, distance_matrix):
    # Calculate the total distance for a given tour (path)
    tour_length = 0
    for i in range(len(tour) - 1):
        tour_length += distance_matrix[tour[i]][tour[i + 1]]
    tour_length += distance_matrix[tour[-1]][tour[0]]  # Return to the start point
    return -tour_length  # We return the negative length to maximize fitness (i.e., minimize distance)

# -------------------- Generate Random Chromosome --------------------
def generate_random_chromosome(chromosome_size):
    return random.sample(range(chromosome_size), chromosome_size)

# -------------------- Crossover (Partially Matched Crossover) --------------------
def crossover(parent1, parent2):
    # PMX crossover
    size = len(parent1)
    child1 = [-1] * size
    child2 = [-1] * size

    # Random crossover points
    cxpoint1 = random.randint(0, size // 2)
    cxpoint2 = random.randint(cxpoint1 + 1, size)

    # Copy the slice from parents
    child1[cxpoint1:cxpoint2] = parent1[cxpoint1:cxpoint2]
    child2[cxpoint1:cxpoint2] = parent2[cxpoint1:cxpoint2]

    # Fill remaining positions
    for i in range(size):
        if parent2[i] not in child1:
            child1[child1.index(-1)] = parent2[i]
        if parent1[i] not in child2:
            child2[child2.index(-1)] = parent1[i]

    return child1, child2

# -------------------- Mutation (Swap Mutation) --------------------
def mutate(chromosome, mutation_rate=0.1):
    if random.random() < mutation_rate:
        # Swap two cities in the tour
        i, j = random.sample(range(len(chromosome)), 2)
        chromosome[i], chromosome[j] = chromosome[j], chromosome[i]
    return chromosome
```

```
# -------------------- Selection (Tournament Selection) --------------------
def select_parents(fitness_list, population_size):
    # Tournament selection: Select the best individual from a random group
    parents = []
    for _ in range(population_size):
        tournament = random.sample(fitness_list, 2)
        tournament.sort(key=lambda x: x[1], reverse=True)  # Sort by fitness (maximizing)
        parents.append(tournament[0][0])
    return parents

# -------------------- Genetic Algorithm for TSP --------------------
def GEA_Optimization(objective_function, population_size, chromosome_size, distance_matrix,
mutation_rate, crossover_rate, generations):
    population = [generate_random_chromosome(chromosome_size) for _ in range(population_size)]

    best_solution = None
    best_fitness = -float('inf')

    # GA loop
    for generation in range(generations):
        fitness_list = []
        for chromosome in population:
            fitness = objective_function(chromosome, distance_matrix)
            fitness_list.append((chromosome, fitness))

            # Update the best solution and fitness
            if fitness > best_fitness:
                best_fitness = fitness
                best_solution = chromosome

        # Select parents using tournament selection
        mating_pool = select_parents(fitness_list, population_size)

        # Generate offspring via crossover and mutation
        offspring = []
        for i in range(0, len(mating_pool), 2):
            parent1 = mating_pool[i]
            parent2 = mating_pool[i + 1]

            # Perform crossover
            if random.random() < crossover_rate:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1.copy(), parent2.copy()

            # Apply mutation
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
```

```
            offspring.append(child1)
            offspring.append(child2)

        # Replace old population with offspring
        population = offspring

        # Print progress every 10 generations
        if generation % 10 == 0:
            print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best Solution =
{best_solution}")

    return best_solution, best_fitness

# ---------- Example Usage ----------
# Define a distance matrix for 4 cities
distance_matrix = np.array([
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
])

# Define parameters for the genetic algorithm
population_size = 50  # Increased population size
chromosome_size = len(distance_matrix)  # Number of cities
mutation_rate = 0.1  # Increased mutation rate
crossover_rate = 0.8  # Increased crossover rate
generations = 100  # Number of generations

# Run the genetic algorithm
best_solution, best_fitness = GEA_Optimization(
    objective_function,
    population_size,
    chromosome_size,
    distance_matrix,
    mutation_rate,
    crossover_rate,
    generations
)

# Output the final best solution
print(f"\nBest Solution Found: {best_solution}")
print(f"Best Fitness (Negative Tour Length): {best_fitness}")
```

Program 3

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.

Implementation Steps:

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

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

3. Initialize Particles: Generate an initial population of particles with random positions and velocities.

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

5. Update Velocities and 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.

Algorithm:

LAB-04

11/9/25

Particle swarm optimization (PSO)

Algorithm:

PSO (function f(x), size N, initialize weight w,
acceleration const c1, c2, max_iteration)
initialize:
for each particle in size N:
Randomly initialize position x_i
Randomly initialize velocity v_i
p_best_i = x_i.

g_best = best among all pbest_i.

loop: for i = 1 to max_iteration:
for particle i in 1:N:
fitness = f(x_i)

if fitness better than (pbest_i):
pbest_i = x_i

if fitness better than (gbest
gbest = x_i.

img pre

for each particle in 1:N:
v_i = w * v_i +
c1 * rand(0,1) * (pbest_i - x_i) *
c2 * rand(0,1) * (gbest - x_i)

x_i = x_i + v_i.

end of loop
output: gbest (best solution found).

Output:-

Iteration 1/50 - Best Threshold : 105.51 - Score : 4648.9937
Iteration 2/50 - Best Threshold : 104.12 - Score : 4648.9937
Iteration 3/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 4/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 5/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 6/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 7/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 8/50 - Best Threshold : 103.12 - Score : 4648.9940

.
.
.
.
.

Iteration 49/50 - Best Threshold : 103.12 - Score : 4648.9940
Iteration 50/50 - Best Threshold : 103.12 - Score : 4648.9940

Code:

```
import numpy as np
import matplotlib.pyplot as plt
from skimage import data
import cv2

# Load the standard grayscale 'Cameraman' image from skimage
image = data.camera()

# Define fitness function - between-class variance for threshold
def between_class_variance(threshold, img):
    threshold = int(threshold)
    hist = np.bincount(img.flatten(), minlength=256)
    total_pixels = img.size

    weight_background = np.sum(hist[:threshold]) / total_pixels
    weight_foreground = np.sum(hist[threshold:]) / total_pixels

    if weight_background == 0 or weight_foreground == 0:
        return 0

    mean_background = np.sum(np.arange(0, threshold) * hist[:threshold]) / np.sum(hist[:threshold])
    mean_foreground = np.sum(np.arange(threshold, 256) * hist[threshold:]) / np.sum(hist[threshold:])

    bc_variance = weight_background * weight_foreground * (mean_background - mean_foreground)
** 2
    return bc_variance

# PSO parameters
num_particles = 30
max_iterations = 50
w = 0.7
c1 = 1.5
c2 = 1.5

# Initialize particles (threshold values 0-255)
positions = np.random.uniform(0, 255, num_particles)
velocities = np.random.uniform(-10, 10, num_particles)

personal_best_positions = positions.copy()
personal_best_scores = np.array([between_class_variance(p, image) for p in positions])

best_particle_idx = np.argmax(personal_best_scores)
global_best_position = personal_best_positions[best_particle_idx]
global_best_score = personal_best_scores[best_particle_idx]

for iteration in range(max_iterations):
    for i in range(num_particles):
```

```
        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]))

        positions[i] += velocities[i]
        positions[i] = np.clip(positions[i], 0, 255)

        fitness = between_class_variance(positions[i], image)

        if fitness > personal_best_scores[i]:
            personal_best_scores[i] = fitness
            personal_best_positions[i] = positions[i]

            if fitness > global_best_score:
                global_best_score = fitness
                global_best_position = positions[i]

    print(f"Iteration {iteration+1}/{max_iterations} — Best Threshold: {global_best_position:.2f} — Score: {global_best_score:.4f}")

best_threshold = int(global_best_position)
_, thresholded_img = cv2.threshold(image, best_threshold, 255, cv2.THRESH_BINARY)

plt.figure(figsize=(12,6))
plt.subplot(1,2,1)
plt.title('Original Image (Cameraman)')
plt.imshow(image, cmap='gray')
plt.axis('off')

plt.subplot(1,2,2)
plt.title(f'PSO Thresholded Image\nThreshold = {best_threshold}')
plt.imshow(thresholded_img, cmap='gray')
plt.axis('off')

plt.show()
```

Program 4

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.

Implementation Steps:

1. Define the Problem: Create a set of cities with their coordinates.

2. Initialize Parameters: Set the number of ants, the importance of pheromone (alpha), the importance of heuristic information (beta), the evaporation rate (rho), and the initial pheromone value.

3. Construct Solutions: Each ant constructs a solution by probabilistically choosing the next city based on pheromone trails and heuristic information.

4. Update Pheromones: After all ants have constructed their solutions, update the pheromone trails based on the quality of the solutions found.

5. Iterate: Repeat the construction and updating process for a fixed number of iterations or until convergence criteria are met.

6. Output the Best Solution: Keep track of and output the best solution found during the iterations.

Algorithm:

LAB-05

Ant colony optimization (ACO) for TSP

Algorithm:

initialize problem value τ on all solution components

int parameter α [pheromone influence], β [heuristic influence], evaporation rate ρ, number of ants m, defination d.

while defination not reached:
  for each ant k in 1 to m:
    initialise an empty solution S_K
    while solution S_K is incomplete:
      select next solution component based on probability proportional to:
      $\tau(c)^{\alpha} * (\eta(c))^{\beta}$
      where $\eta(c)$ is heuristic desirability of component c. and component c to solution S_K evaluate the quality of solution S_K.

  for each solution component c:
    evaporate pheromone:
    $\tau(c) = (1-\rho)^{*} \tau(c)$

  for each ant k:
    deposit pheromone on component in S_K:
    $\tau(c) = \tau(c) + \Delta\tau\_K(c)$
    amount $\Delta\tau\_K(c)$ depends on quality of solution S_K.
return the best solution found.

N/w → Packets transmission via routers.
Observe: 1) Packets drop chages
2) Packets traffic

Output:-

Scenario 1: low packet drops, low traffic ::

Iteration 1 | Best path length : 15.67 | packet drops : 17/20 |
Traffic load : 1

Iteration 2 | Best path length : 15.67 | packet drops : 8/20 |
Traffic load : 1

⋮

Iteration 49 | Best path length : 15.67 | packet drops : 1/20 |
Traffic load : 1

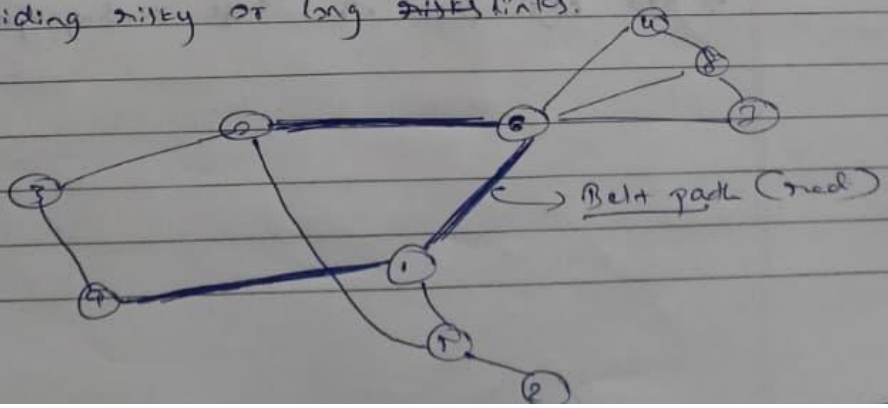
Iteration 50 | Best path length : 15.67 | packet drops : 2/20 |
Traffic load : 1

Network Best path found by ACO.

Observation

-> packet drops are rare (<5% chance per link).
-> most ants successfully reach the destination.
-> The best path length tends to decrease smoothly over iterations as ants converge on optimal routes.
-> Because traffic is low, system explores without congestion.

The best path will be relatively short, reliable path avoiding risky or long ~~links~~ links.



Scenario 2: higher packet drop:

Iteration 1 | Best path length : 1.02 | packet drops: 13/20 | Traffic load : 1

Iteration 2 | Best path length : 1.02 | packet drops : 9/20 | Traffic load : 1

:

:

Iteration 49/50 | Best path length : 1.02 | packet drops : 6/20 | Traffic load : 1

Iteration. 50 | Best path length : 1.02 | packet drops : 8/20 | Traffic load : 1.

Observation

-> Around 30% packet drop rate means many ants fail to reach the destination

-> pheromone updates are less frequent and more concentrated on reliable paths.

-> Best path may be higher initially due to detours avoiding lossy links.

-> Best path highlighted may be longer but more reliable.

-> Red edges show the "safe routing path

Scenario 3: Increased traffic load.

Iteration 1 | Best path length: 8.68 | packet drops: 94100 | Traffic load: 5

Iteration 2 | Best path length: 8.68 | packet drops: 5100 | Traffic load: 5

Iteration 49 | Best path length: 8.68 | packet drops: 6100 | Traffic load: 5

Iteration 50 | Best path length: 8.68 | packet drops: 9100 | Traffic load: 5

Observation

-> Traffic is 5x higher, so many ants explore routes every iteration.

-> faster pheromone updates lead to quicker convergence on good paths.

-> The best path highlighted is similar to Scenario 1 but founder faster.

-> Network topology is the same but pheromone strength on best path edges is higher.

Code:

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

# Create a graph representing routers and links
def create_network(num_nodes=10, connection_prob=0.3):
    G = nx.erdos_renyi_graph(num_nodes, connection_prob, directed=False)
    while not nx.is_connected(G):
        G = nx.erdos_renyi_graph(num_nodes, connection_prob, directed=False)
    return G

# Initialize pheromone matrix
def init_pheromone(G, initial_pheromone=1.0):
    pheromone = {}
    for edge in G.edges():
        pheromone[edge] = initial_pheromone
        pheromone[(edge[1], edge[0])] = initial_pheromone  # undirected graph
    return pheromone

# Probability to choose next hop based on pheromone and heuristic (inverse of link cost)
def choose_next_hop(current_node, unvisited, G, pheromone, alpha, beta):
    pheromone_list = []
    heuristic_list = []
    neighbors = list(G.neighbors(current_node))
    feasible_neighbors = [n for n in neighbors if n in unvisited]
    if not feasible_neighbors:
        return None

    for neighbor in feasible_neighbors:
        tau = pheromone[(current_node, neighbor)] ** alpha
        eta = (1 / G[current_node][neighbor]['weight']) ** beta
        pheromone_list.append(tau)
        heuristic_list.append(eta)

    probabilities = np.array(pheromone_list) * np.array(heuristic_list)
    probabilities /= probabilities.sum()
    next_node = np.random.choice(feasible_neighbors, p=probabilities)
    return next_node

# Simulate packet routing by ants (packets)
def ant_routing(G, pheromone, alpha, beta, source, destination, packet_drop_rate=0.0):
    path = [source]
    current = source
    visited = set([source])
    while current != destination:
        neighbors = list(G.neighbors(current))
```

```
        feasible = [n for n in neighbors if n not in visited]
        if not feasible:
            # Dead end or loop, restart from source or break
            return None  # packet dropped or no route
        next_node = choose_next_hop(current, feasible, G, pheromone, alpha, beta)
        # Simulate packet drop
        if random.random() < packet_drop_rate:
            return None  # packet dropped on this edge
        path.append(next_node)
        visited.add(next_node)
        current = next_node
    return path

# Update pheromones with evaporation and reinforcement
def update_pheromones(G, pheromone, paths, rho, Q=100):
    # Evaporation
    for edge in pheromone:
        pheromone[edge] *= (1 - rho)
        if pheromone[edge] < 0.01:
            pheromone[edge] = 0.01  # minimum pheromone

    # Deposit pheromones for successful paths
    for path in paths:
        if path is None:
            continue
        length = sum(G[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
        deposit = Q / length
        for i in range(len(path)-1):
            edge = (path[i], path[i+1])
            pheromone[edge] += deposit
            pheromone[(edge[1], edge[0])] += deposit  # undirected

# Run simulation
def run_simulation(num_nodes=10, num_ants=20, alpha=1, beta=2, rho=0.1,
                   packet_drop_rate=0.0, traffic_load=1, iterations=50):
    G = create_network(num_nodes)
    # Assign random weights (delay or cost) to edges
    for u, v in G.edges():
        G[u][v]['weight'] = random.uniform(1, 10)

    pheromone = init_pheromone(G)
    source, destination = 0, num_nodes - 1  # fixed source and destination

    best_path = None
    best_length = float('inf')

    for iteration in range(iterations):
        all_paths = []
        for _ in range(num_ants * traffic_load):  # traffic load scales ants count
```

```
            path = ant_routing(G, pheromone, alpha, beta, source, destination, packet_drop_rate)
            all_paths.append(path)

        update_pheromones(G, pheromone, all_paths, rho)

        # Track best path found this iteration
        for path in all_paths:
            if path is None:
                continue
            length = sum(G[path[i]][path[i+1]]['weight'] for i in range(len(path)-1))
            if length < best_length:
                best_length = length
                best_path = path

        print(f"Iteration {iteration+1} | Best path length: {best_length:.2f} | Packet drops: {sum(p is None for p in all_paths)} / {len(all_paths)} | Traffic load: {traffic_load}")

    # Plot network and best path
    pos = nx.spring_layout(G)
    plt.figure(figsize=(8,6))
    nx.draw(G, pos, with_labels=True, node_color='lightblue')
    if best_path:
        path_edges = list(zip(best_path[:-1], best_path[1:]))
        nx.draw_networkx_edges(G, pos, edgelist=path_edges, edge_color='r', width=2)
    plt.title('Network and Best Path Found by ACO')
    plt.show()

    return best_path, best_length

# Example runs:

print("=== Scenario 1: Low packet drop, low traffic ===")
run_simulation(packet_drop_rate=0.05, traffic_load=1)

print("\n=== Scenario 2: Higher packet drop ===")
run_simulation(packet_drop_rate=0.3, traffic_load=1)

print("\n=== Scenario 3: Increased traffic load ===")
run_simulation(packet_drop_rate=0.05, traffic_load=5)
```

Program 5

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.

Implementation Steps:

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

2. Initialize Parameters: Set the number of nests, the probability of discovery, and the number of iterations.

3. Initialize Population: Generate an initial population of nests with random positions.

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

5. Generate New Solutions: Create new solutions via Lévy flights.

6. Abandon Worst Nests: Abandon a fraction of the worst nests and replace them with new random positions.

7. Iterate: Repeat the evaluation, updating, and replacement process for a fixed number of iterations or until convergence criteria are met.

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

Algorithm:

LAB - 06

16-10-25

Cuckoo Search (CS):-

Step-1) Initialize parameters
n: number of nests (population size)
pa: discovery rate of alien eggs (typically 0.25)
α: step size scaling factor
max Gen: maximum number of generations.

Step-2) generate initial population of n host nests x_i ($i = 1, 2, 3, \ldots n$)

Step-3) evaluate fitness $f(x_i)$ for all nests.

Step-4) while ($t <$ max Gen) do:
a. generate a new solution x_i' for a cuckoo by levy flight.

$$x_i' = x_i + \alpha * \text{levy}(\lambda)$$

b. Evaluate fitness $f(x_i')$
c. Randomly select a nest j among n
d. If $f(x_i') > f(x_j)$:
Replace x_j with x_i'
e. Abandon a fraction pa of worst nests & replace them with new random solutions.

f. Keep the best solution (nest)

g. Rank the solutions & find the current Best. $t = t + 1$

Step 5:- Output the best solution found.

Output:

Application :- Smart Home Energy optimization

Enter the number of devices : 2

Enter details for device 1 :
Device name : washing machine
How many hours does washing machine need to run : 2
earliest start hour (0-23) : 18
latest end hour (0-23) : 24
power consumption (KWS) : 15

Enter details for device 2 :
Device name : EV charger
how many hours does EV charger need to run : 3
earliest start hour (0-23) : 16
latest end hour (0-23) : 24
power consumption (KWH) : 36.

optimal Schedule found :
washing machine : start hour 20, end at 22
EV charger : start hour 19, End at 22

Minimum total energy cost = 48.00 units.

Code:

```
import numpy as np
import math  # Import Python's math module

# Example processing times for n jobs
processing_times = np.array([2, 5, 1, 3, 4])  # Change this as needed
n_jobs = len(processing_times)

# Objective function: sum of completion times (total completion time)
def total_completion_time(order):
    completion_time = 0
    current_time = 0
    for job in order:
        current_time += processing_times[job]
        completion_time += current_time
    return completion_time

# Convert continuous vector to job permutation (random-key encoding)
def vector_to_permutation(vec):
    return np.argsort(vec)

# Levy flight step generator
def levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)

    u = np.random.normal(0, sigma, size=dim)
    v = np.random.normal(0, 1, size=dim)

    step = u / np.abs(v) ** (1 / Lambda)
    return step

# Parameters
n_nests = 25           # Number of nests (solutions)
pa = 0.25              # Probability of abandoning worse nests
n_iterations = 1000    # Number of iterations
dim = n_jobs           # Dimension of the problem (number of jobs)

# Initialize nests randomly in [0,1]
nests = np.random.rand(n_nests, dim)
fitness = np.array([total_completion_time(vector_to_permutation(nests[i]))
                    for i in range(n_nests)])

# Find the best initial solution
best_idx = np.argmin(fitness)
best_nest = nests[best_idx].copy()
best_fitness = fitness[best_idx]
```

```
for t in range(n_iterations):

    # Generate new solutions via Levy flights
    for i in range(n_nests):
        step = levy_flight(1.5, dim)
        new_solution = nests[i] + 0.01 * step
        new_solution = np.clip(new_solution, 0, 1)  # Keep within [0,1]

        new_fitness = total_completion_time(vector_to_permutation(new_solution))

        # Replace if better
        if new_fitness < fitness[i]:
            nests[i] = new_solution
            fitness[i] = new_fitness

            if new_fitness < best_fitness:
                best_fitness = new_fitness
                best_nest = new_solution.copy()

    # Abandon some nests and generate new ones
    K = np.random.rand(n_nests, dim) > pa

    # Use a consistent permutation for difference calculation
    perm = np.random.permutation(n_nests)
    stepsize = np.random.rand(n_nests, dim) * (nests[perm] - nests[np.roll(perm, 1)])

    nests = nests + stepsize * K
    nests = np.clip(nests, 0, 1)

    # Evaluate new nests
    for i in range(n_nests):
        new_fitness = total_completion_time(vector_to_permutation(nests[i]))

        if new_fitness < fitness[i]:
            fitness[i] = new_fitness

        if new_fitness < best_fitness:
            best_nest = nests[i].copy()
            best_fitness = new_fitness

    # Print progress every 100 iterations
    if (t + 1) % 100 == 0:
        print(f"Iteration {t + 1}, Best total completion time: {best_fitness}")

print("\nOptimization finished.")
print("Best job order:", vector_to_permutation(best_nest))
print("Best total completion time:", best_fitness)
```

Program 6

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.

Implementation Steps:

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

2. Initialize Parameters: Set the number of wolves and the number of iterations.

3. Initialize Population: Generate an initial population of wolves with random positions.

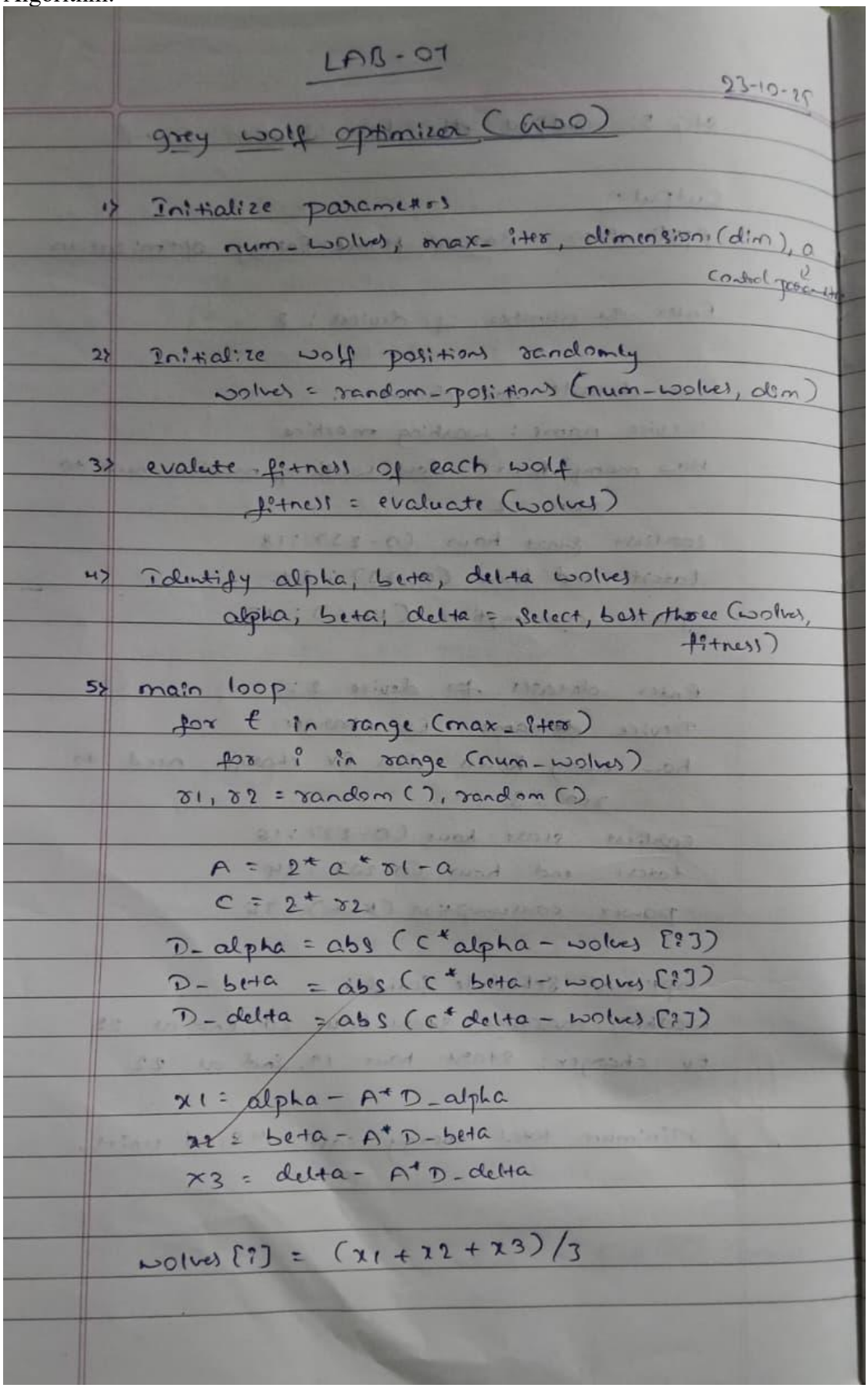
4. Evaluate Fitness: Evaluate the fitness of each wolf based on the optimization function.

5. Update Positions: Update the positions of the wolves based on the positions of alpha, beta, and delta wolves.

6. Iterate: Repeat the evaluation and position updating process 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.

Algorithm:



LAB - 07

23-10-25

grey wolf optimizer (GWO)

1) Initialize parameters
num_wolves, max_iter, dimension (dim), a
control parameter

2) Initialize wolf positions randomly
wolves = random_positions (num_wolves, dim)

3) evalute fitness of each wolf
fitness = evaluate (wolves)

4) Identify alpha, beta, delta wolves
alpha, beta, delta = select_best_three (wolves, fitness)

5) main loop
for t in range (max_iter)
for i in range (num_wolves)
r1, r2 = random (), random ()

A = 2 * a * r1 - a
C = 2 * r2
D_alpha = abs (C * alpha - wolves [i])
D_beta = abs (C * beta - wolves [i])
D_delta = abs (C * delta - wolves [i])

x1 = alpha - A * D_alpha
x2 = beta - A * D_beta
x3 = delta - A * D_delta

wolves [i] = (x1 + x2 + x3) / 3

fitness = evaluate (wolves)

alpha, beta, delta = select_best_three (wolves, fitness)

a = 2 - t * (2 / max_iter)

6) Return best solution.

best_solution = alpha

print ("Best Solution", best_solution)

o/p Application :- Image processing.

Iteration 1/20, Best fitness : 0.0047

Iteration 2/20, Best fitness : 0.0047

Iteration 3/20, Best fitness : 0.0059

Iteration 4/20, Best fitness : 0.0060

Iteration 5/20, Best fitness : 0.0060

Iteration 6/20, Best fitness : 0.0081

Iteration 7/20, Best fitness : 0.0175

Iteration 8/20, Best fitness : 0.0175

Iteration 16/20, Best fitness : 0.0175

Iteration 17/20, Best fitness : 0.0175

Iteration 18/20, Best fitness : 0.0175

Iteration 19/20, Best fitness : 0.0175

Iteration : 20/20, Best fitness : 0.0175

Optimized thresholds : [0, 1.88649006]

fitness : 0.0175

Code:

```
# pip install numpy pandas scikit-learn

import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score


# --------------------------
# Grey Wolf Optimizer (GWO)
# --------------------------

def fitness_function(features, X_train, X_test, y_train, y_test):
    # Select only chosen features
    cols = [i for i in range(len(features)) if features[i] > 0.5]
    if len(cols) == 0:
        return 0

    X_train_sel = X_train[:, cols]
    X_test_sel = X_test[:, cols]

    # Train simple KNN classifier
    clf = KNeighborsClassifier(n_neighbors=3)
    clf.fit(X_train_sel, y_train)
    y_pred = clf.predict(X_test_sel)

    return accuracy_score(y_test, y_pred)


def GWO(X_train, X_test, y_train, y_test, n_wolves=10, max_iter=20):
    dim = X_train.shape[1]

    alpha_pos = np.zeros(dim)
    beta_pos = np.zeros(dim)
    delta_pos = np.zeros(dim)

    alpha_score = beta_score = delta_score = 0

    wolves = np.random.rand(n_wolves, dim)

    for t in range(max_iter):
        for i in range(n_wolves):
            fitness = fitness_function(wolves[i], X_train, X_test, y_train, y_test)

            if fitness > alpha_score:
```

```
            alpha_score, alpha_pos = fitness, wolves[i].copy()

        elif fitness > beta_score:
            beta_score, beta_pos = fitness, wolves[i].copy()

        elif fitness > delta_score:
            delta_score, delta_pos = fitness, wolves[i].copy()

    a = 2 - t * (2 / max_iter)  # Linearly decreasing

    for i in range(n_wolves):
        for j 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[j] - wolves[i][j])
            X1 = alpha_pos[j] - 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[j] - wolves[i][j])
            X2 = beta_pos[j] - 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[j] - wolves[i][j])
            X3 = delta_pos[j] - A3 * D_delta

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

    wolves = np.clip(wolves, 0, 1)

    print(f"Iteration {t+1}/{max_iter}, Best Accuracy: {alpha_score:.4f}")

return alpha_pos, alpha_score


# --------------------------
# Run GWO on Iris dataset
# --------------------------

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

best_features, best_acc = GWO(X_train, X_test, y_train, y_test)
```

```
selected_features = [i for i, val in enumerate(best_features) if val > 0.5]

print("\nBest selected features:", selected_features)
print("Best Accuracy:", best_acc)
```

Program 7

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.

Implementation Steps:

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

2. Initialize Parameters: Set the number of cells, grid size, neighborhood structure, and number of iterations.

3. Initialize Population: Generate an initial population of cells with random positions in the solution space.

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

5. Update States: Update the state of each cell based on the states of its neighboring cells and predefined update rules.

6. Iterate: Repeat the evaluation and state updating process 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.

Algorithm:

LAB - 08

30/10/25

Parallel Cellular algorithm

algorithm :=>

1> Start
2> Define the optimization problem :
define a function $f(x)$ to be minimized or maximized
3> Initialize parameters :
Set NUM_CELLS ← number of cells
Set GRID_SIZE ← size of grid
Set MAX_ITER ← maximum number of iterations
Define NEIGHBORHOOD structure (e.g., 4-neighbors or 8-neighbors)
4> Initialize population :
for each cell i in the grid :
Assign a random position x[i] in the solution space
compute fitness [i] = f(x[i])
5> Repeat for ITER = 1 to MAX_ITER :
For each cell i in parallel :
Find neighboring cells based on NEIGHBORHOOD
update state of cells i using update rule :
x[i] ← updateRule (x[i], neighbors)
Recalculate fitness [i] = f(x[i])
Find the best cell with highest (or lowest) fitness
6> until convergence or ITER = MAX_ITER
7> Output : Print the best solution and its fitness value
8> Stop.

Output:-

Initial Best fitness : 0.0668 at [0.2370959 0.1030856 69]

Iteration 10 : Best fitness = 0.0172
Iteration 20 : Best fitness = 0.0000
Iteration 30 : Best fitness = 0.0000
Iteration 40 : Best fitness = 0.0000
Iteration 50 : Best fitness = 0.0000
Iteration 60 : Best fitness = 0.0000
Iteration 70 : Best fitness = 0.0000
Iteration 80 : Best fitness = 0.0000
Iteration 90 : Best fitness = 0.0000
Iteration 100 : Best fitness = 0.0000

Optimization complete.
Final Best Solution (x): [6.30710831e-06 -2.581184 81e-06]

final Best fitness (f(x)): 0.000000.

Explanation (Sphere function).

> The primary goal of minimizing $f(x) = x_1^2 + x_2^2$ is to find x_1 & x_2 that make as close to zero

> Iteration 20 shows the Best fitness = 0.00000. for the rest of the Iterations, confirming stability. (so it remained at zero).

> It indicates that (x) that the cells successfully converged to the center of the search space, where the minimum of the sphere function is located.

Improvement over time

Era Implementation True parallelism.

early (2000) → Theoretical parallel → conceptual only.
CA.

Mid (2010-20) → Multi core/GPU. → yes

Modern (2020-25) → AI-Integrated → fully parallel.
CA.

Time complexity = $O(n^2)$
$O(n^2)$ but faster in practice
$O(n^2)$ real-time performance.

Space complexity = $2 \times n^2$
n^2 (optimized)
n^2 (or less)

Improvement → Basic rules sequential loops.
hardware based parallism.

→ In old papers established the theoretical foundation - proving that complexity, universal computation, & self reproduction could exist in simple, discrete systems.

→ New papers focus on practical application, extending CA model to irregular structures.

Code:

```
import numpy as np
import random

# --- 2. Define the optimization problem: ---
def objective_function(x):
    """The function f(x) to be minimized (e.g., Sphere Function)."""
    # x is a 2D array: [x1, x2]
    return np.sum(x**2)

# --- Define the Update Rule ---
def update_rule(current_position, neighbors_data):
    """
    Basic update rule: Move towards the position of the best neighbor.

    neighbors_data is a list of tuples: [(position, fitness), ...]
    """
    if not neighbors_data:
        # No neighbors (e.g., on the edge and only checking inbounds) - keep position
        return current_position

    # Find the neighbor with the minimum (best) fitness
    best_neighbor = min(neighbors_data, key=lambda item: item[1])
    best_neighbor_pos = best_neighbor[0]

    # Simple update: move a small step towards the best neighbor
    learning_rate = 0.1
    new_position = current_position + learning_rate * (best_neighbor_pos - current_position)

    return new_position

# --- 3. Initialize parameters: ---
NUM_CELLS_DIM = 20          # Cells along one dimension (e.g., 20x20 grid)
NUM_CELLS = NUM_CELLS_DIM * NUM_CELLS_DIM
GRID_SIZE = NUM_CELLS_DIM
MAX_ITER = 100
SEARCH_SPACE_BOUNDS = [-5.0, 5.0]  # Range for x1 and x2
SOLUTION_DIMENSION = 2      # The dimension of the solution vector x (e.g., [x1, x2])

# Define NEIGHBORHOOD structure (8-neighbors: Moore neighborhood)
# Relative coordinates for 8 neighbors
NEIGHBORHOOD = [(-1, -1), (-1, 0), (-1, 1),
                ( 0, -1),          ( 0, 1),
                ( 1, -1), ( 1, 0), ( 1, 1)]

# --- 4. Initialize population: ---
# Initialize positions: x[i] will be a 2D vector [x1, x2]
positions = np.zeros((NUM_CELLS, SOLUTION_DIMENSION))
```

```
fitness = np.zeros(NUM_CELLS)

for i in range(NUM_CELLS):
    # Assign a random position x[i] in the solution space
    low, high = SEARCH_SPACE_BOUNDS
    positions[i] = np.random.uniform(low, high, SOLUTION_DIMENSION)

    # Compute fitness[i] = f(x[i])
    fitness[i] = objective_function(positions[i])

best_global_fitness = np.min(fitness)
best_global_position = positions[np.argmin(fitness)].copy()

print(f"Initial Best Fitness: {best_global_fitness:.4f} at {best_global_position}")
print("-" * 30)

# --- 5. Repeat for ITER = 1 to MAX_ITER: ---
for iter_count in range(1, MAX_ITER + 1):

    new_positions = positions.copy()
    new_fitness = fitness.copy()

    # Simulate parallel update for each cell i
    for i in range(NUM_CELLS):

        # Convert 1D index i to 2D grid coordinates (row, col)
        r = i // GRID_SIZE
        c = i % GRID_SIZE

        # Find neighboring cells based on NEIGHBORHOOD
        neighbors_data = []
        for dr, dc in NEIGHBORHOOD:
            nr, nc = r + dr, c + dc

            # Check bounds
            if 0 <= nr < GRID_SIZE and 0 <= nc < GRID_SIZE:
                neighbor_idx = nr * GRID_SIZE + nc

                neighbor_pos = positions[neighbor_idx]
                neighbor_fitness = fitness[neighbor_idx]

                neighbors_data.append((neighbor_pos, neighbor_fitness))

        # Update state of cell i using update rule:
        new_pos = update_rule(positions[i], neighbors_data)

        # Ensure the new position stays within the defined search space bounds
        low, high = SEARCH_SPACE_BOUNDS
        new_positions[i] = np.clip(new_pos, low, high)
```

```
            # Recalculate fitness[i] = f(x[i])
            new_fitness[i] = objective_function(new_positions[i])

        # Update the entire population for the next iteration
        positions = new_positions
        fitness = new_fitness

        # Find the best cell with highest (or lowest) fitness
        current_best_fitness = np.min(fitness)

        # Update global best
        if current_best_fitness < best_global_fitness:
            best_global_fitness = current_best_fitness
            best_global_position = positions[np.argmin(fitness)].copy()

        # --- 6. Until convergence or ITER = MAX_ITER ---
        # (Simplified: just print status every 10 iterations)
        if iter_count % 10 == 0 or iter_count == MAX_ITER:
            print(f"Iteration {iter_count:3d}: Best Fitness = {best_global_fitness:.4f}")

print("-" * 30)

# --- 7. Output: ---
print("\n☑ Optimization Complete.")
print(f"Final Best Solution (x): {best_global_position}")
print(f"Final Best Fitness (f(x)): {best_global_fitness:.8f}")
```