VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Chethan K S
USN: 1BM23CS074

in partial fulfilment 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 Dec-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 **Chethan K S (1BM23CS074),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

Raghavendra C K Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

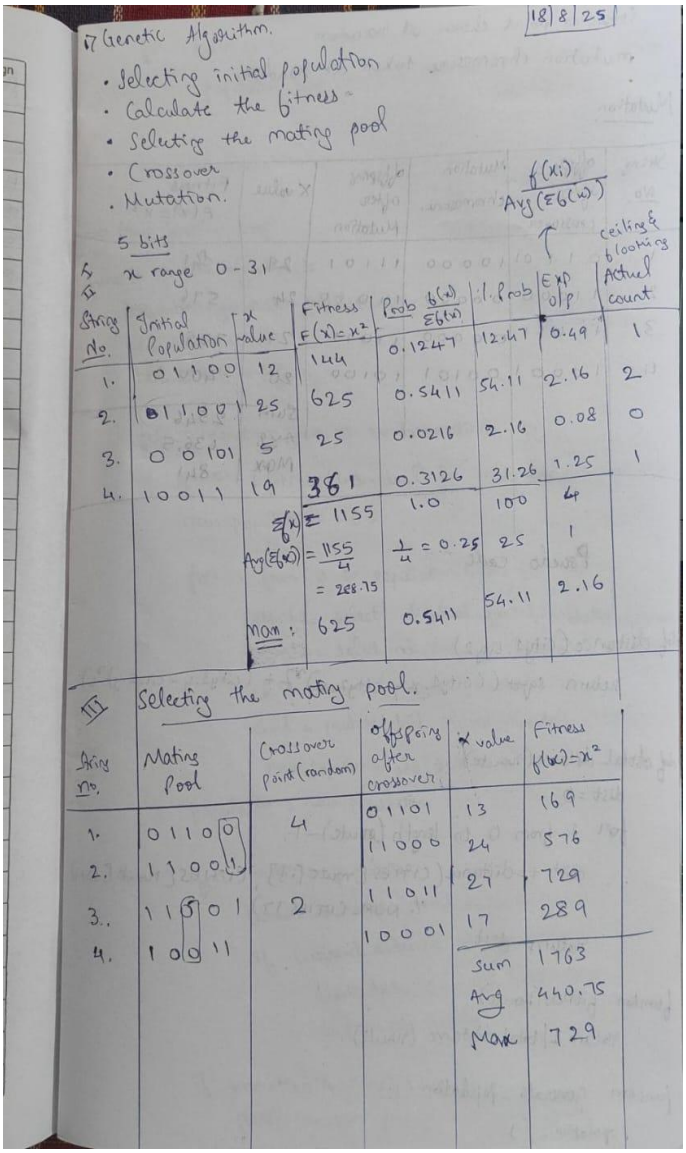
Index

GitHub Link:
https://github.com/Chethan-K-S/BIS_SEM5

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.

Algorithm:



i) Genetic Algorithm. 18/8/25

- Selecting initial population
- Calculate the fitness
- Selecting the mating pool
- Crossover
- Mutation.

5 bits

x range 0 - 31

$\frac{f(x_i)}{Avg(\Sigma f(x))}$ ↑ ceiling & flooring

String No.	Initial Population	x value	Fitness $F(x)=x^2$	Prob $\frac{f(x)}{\Sigma f(x)}$	% Prob	Exp o/p	Actual count
1.	01100	12	144	0.1247	12.47	0.49	1
2.	11001	25	625	0.5411	54.11	2.16	2
3.	00101	5	25	0.0216	2.16	0.08	0
4.	10011	19	361	0.3126	31.26	1.25	1
			$\Sigma f(x) = 1155$	1.0	100	4	
			Avg$(\Sigma f(x)) = \frac{1155}{4}$	$\frac{1}{4} = 0.25$	25	1	
			$= 288.75$		54.11	2.16	
		Max:	625	0.5411			

III Selecting the mating pool.

String no.	Mating Pool	Crossover Point (random)	offspring after crossover	x value	Fitness $f(x)=x^2$
1.	0110 | 0	4	01101	13	169
2.	1100 | 1		11000	24	576
3.	11 | 0 1	2	11011	27	729
4.	10 | 0 11		10001	17	289
				Sum	1763
				Avg	440.75
				Max	729

Crossover point chosen at random
mutation chromosome taken at random.

Mutation

String No.	offspring after crossover	Mutation chromosome	offspring after Mutation	x value	Fitness $f(x) = x^2$
1.	0 1 1 0 1	1 0 0 0 0	1 1 0 0 1	29	841
2.	1 1 0 0 0	0 0 0 0 0	1 1 0 0 0	24	576
3.	1 1 0 1 1	0 0 0 0 0	1 1 0 1 1	27	729
4.	1 0 0 0 1	0 0 1 0 1	1 0 1 0 0	20	400
				Sum	2546
				Avg	636.5
				Max	841

Pseudo code.

```
def distance(city1, city2)
    return sqrt((city1.x - city2.x)^2 + (city1.y - city2.y)^2)

def total_distance(route)
    dist = 0
    for i from 0 to length(route) - 1:
        dist += distance(CITIES[route[i]], CITIES[route[(i+1) % NUM_CITIES]])
    return dist

function fitness(route)
    return 1 / total_distance(route)

function generate_population():
    population = []
    for i from 0 to population_size - 1:
        population.add(random permutation of cities from 0 to num_cities-1)
    return population
```

⇒ Main genetic loop

```
function genetic_algorithm():
    population = generate_population()
    best_route = null
    best_distance = infinity

    for generation from 0 to Generations - 1:
        fitness = [fitness(individual) for individual in population]
        new_population = []

        for i from 0 to population_size - 1:
            parent1 = select_individual_from(population, fitness)
            parent2 = select_individual_from(population, fitness)
            child = crossover(parent1, parent2)
            child = mutate(child, mutation_rate)
            new_population.add(child)
        population = new_population

        for route in population:
            current_distance = total_distance(route)
            if current_distance < best_distance:
                best_distance = current_distance
                best_route = route

        If generation % 50 == 0:
            print("Generation", generation, ": Best Distance =",
                  round(best_distance, 2))
```

⇒ Crossover function

```
function Crossover(parent1, parent2):
    start, end = random indices
    child = empty list
    child[start:end] = parent1[start:end]

    for gene in parent2:
        if gene is not in child:
            insert gene into child at next empty spot
    return child

function mutate(route, mutation_rate):
    if random() < mutation_rate:
        i, j = random indices
        Swap route[i] and route[j]
    return route.
```

output

```
Generation 0 : Best Distance = 278.47
Generation 50 : Best Distance = 251.36
Generation 100 : Best Distance = 242.07
    .
    .
    .
250 :  = 242.07
300 :  = 241.81
    .
    .
    .
Generation 450 : Best Distance = 241.81

Final Best Route :
Route: [5, 4, 2, 0, 9, 3, 8, 1, 7, 6]
Distance : 241.81
```

Output trace 5 cities.

```
Population_size = 100
Generations = 1
mutation rate = 0.1
Cities = [(81, 93), (41, 19), (65, 43), (28, 77), (13, 11)]
Population of 100 random routes (permutation of 0 to 4)
Ex  [2, 0, 4, 1, 3]
```

first generation
total dist.
fitness 0 calculated
Selection → 100 parents → higher probability - higher fitness
Crossover → random slice from parent 1 Eg fills with genes from parent 2.
Mutation 10% chance
Update population
→ Track best route
→ Iterates all the 100 new routes to find lowest total distance.
→ Best distance.

Code:

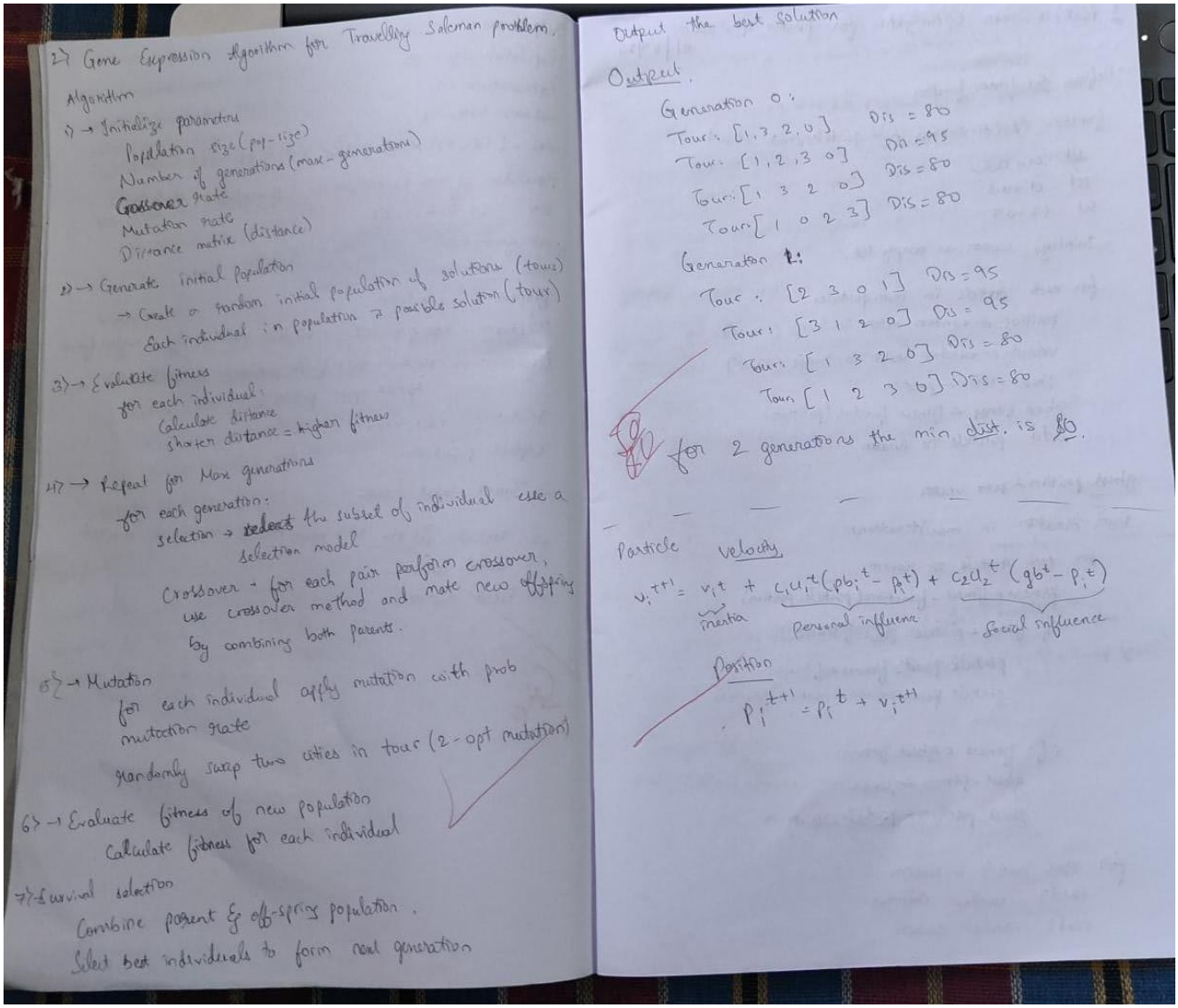
```
import random
import math

NUM_CITIES = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1

cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(NUM_CITIES)]

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

def total_distance(route):
    dist = 0
    for i in range(len(route)):
        dist += distance(cities[route[i]], cities[route[(i + 1) % NUM_CITIES]])
    return dist

def fitness(route):
    return 1 / total_distance(route)

def generate_population():
    return [random.sample(range(NUM_CITIES), NUM_CITIES) for _ in range(POPULATION_SIZE)]

def selection(population, fitnesses):
    selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
    return selected

def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES
    child[start:end] = parent1[start:end]
    pointer = 0
    for gene in parent2:
        if gene not in child:
            while child[pointer] is not None:
                pointer += 1
            child[pointer] = gene
    return child

def mutate(route):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
        route[i], route[j] = route[j], route[i]
    return route
```

```
def genetic_algorithm():
    population = generate_population()
    best_route = None
    best_distance = float('inf')

    for generation in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
        new_population = []

        for _ in range(POPULATION_SIZE):
            parent1, parent2 = selection(population, fitnesses)[:2]
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        population = new_population

        for route in population:
            dist = total_distance(route)
            if dist < best_distance:
                best_distance = dist
                best_route = route

        if generation % 50 == 0:
            print(f"Generation {generation}: Best Distance = {round(best_distance, 2)}")

    print("\n Final Best Route:")
    print("Route:", best_route)
    print("Distance:", round(best_distance, 2))

genetic_algorithm()
```

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.

Algorithm:



Code:

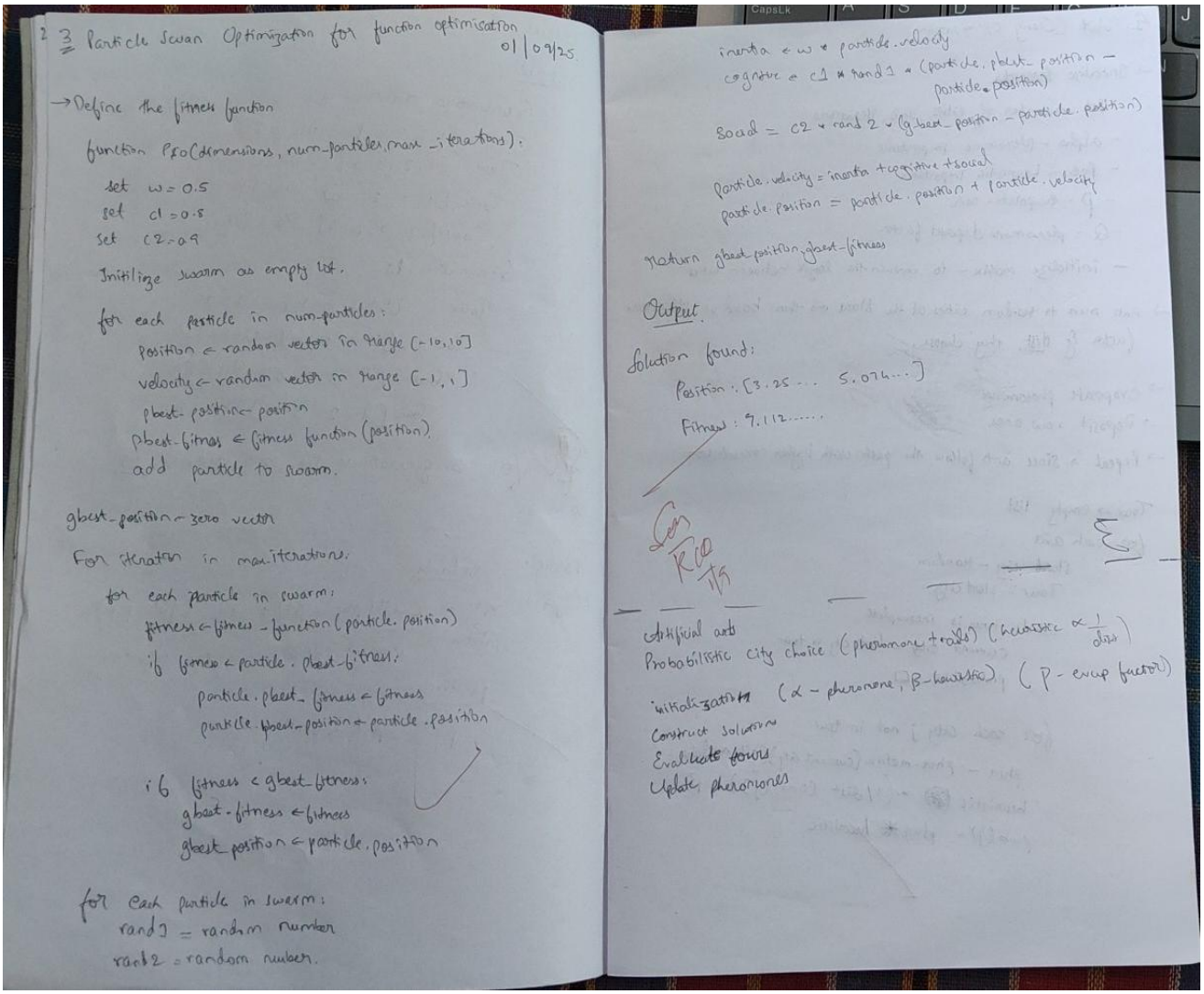
```
import random
import math

# Parameters
NUM_CITIES = 10
POPULATION_SIZE = 100
GENERATIONS = 500
MUTATION_RATE = 0.1

# Generate random cities
cities = [(random.randint(0, 100), random.randint(0, 100)) for _ in range(NUM_CITIES)]

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

def total_distance(route):
    dist = 0
    for i in range(len(route)):
        dist += distance(cities[route[i]], cities[route[(i + 1) % NUM_CITIES]])
    return dist

def fitness(route):
    return 1 / total_distance(route)

def generate_population():
    return [random.sample(range(NUM_CITIES), NUM_CITIES) for _ in range(POPULATION_SIZE)]

def selection(population, fitnesses):
    selected = random.choices(population, weights=fitnesses, k=POPULATION_SIZE)
    return selected

def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(NUM_CITIES), 2))
    child = [None] * NUM_CITIES
    child[start:end] = parent1[start:end]
    pointer = 0
    for gene in parent2:
        if gene not in child:
            while child[pointer] is not None:
                pointer += 1
            child[pointer] = gene
    return child

def mutate(route):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(NUM_CITIES), 2)
        route[i], route[j] = route[j], route[i]
    return route
```

```
def genetic_algorithm():
    population = generate_population()
    best_route = None
    best_distance = float('inf')

    for generation in range(GENERATIONS):
        fitnesses = [fitness(ind) for ind in population]
        new_population = []

        for _ in range(POPULATION_SIZE):
            parent1, parent2 = selection(population, fitnesses)[:2]
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        population = new_population

        # Track best
        for route in population:
            dist = total_distance(route)
            if dist < best_distance:
                best_distance = dist
                best_route = route

        if generation % 50 == 0:
            print(f"Generation {generation}: Best Distance = {round(best_distance, 2)}")

    print("\n🏁 Final Best Route:")
    print("Route:", best_route)
    print("Distance:", round(best_distance, 2))

genetic_algorithm()
```

Program 3: Particle Swarm Optimization for Function Optimization:

Particle Swarm Optimization (PSO) is inspired by the social behaviour 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:



Code:

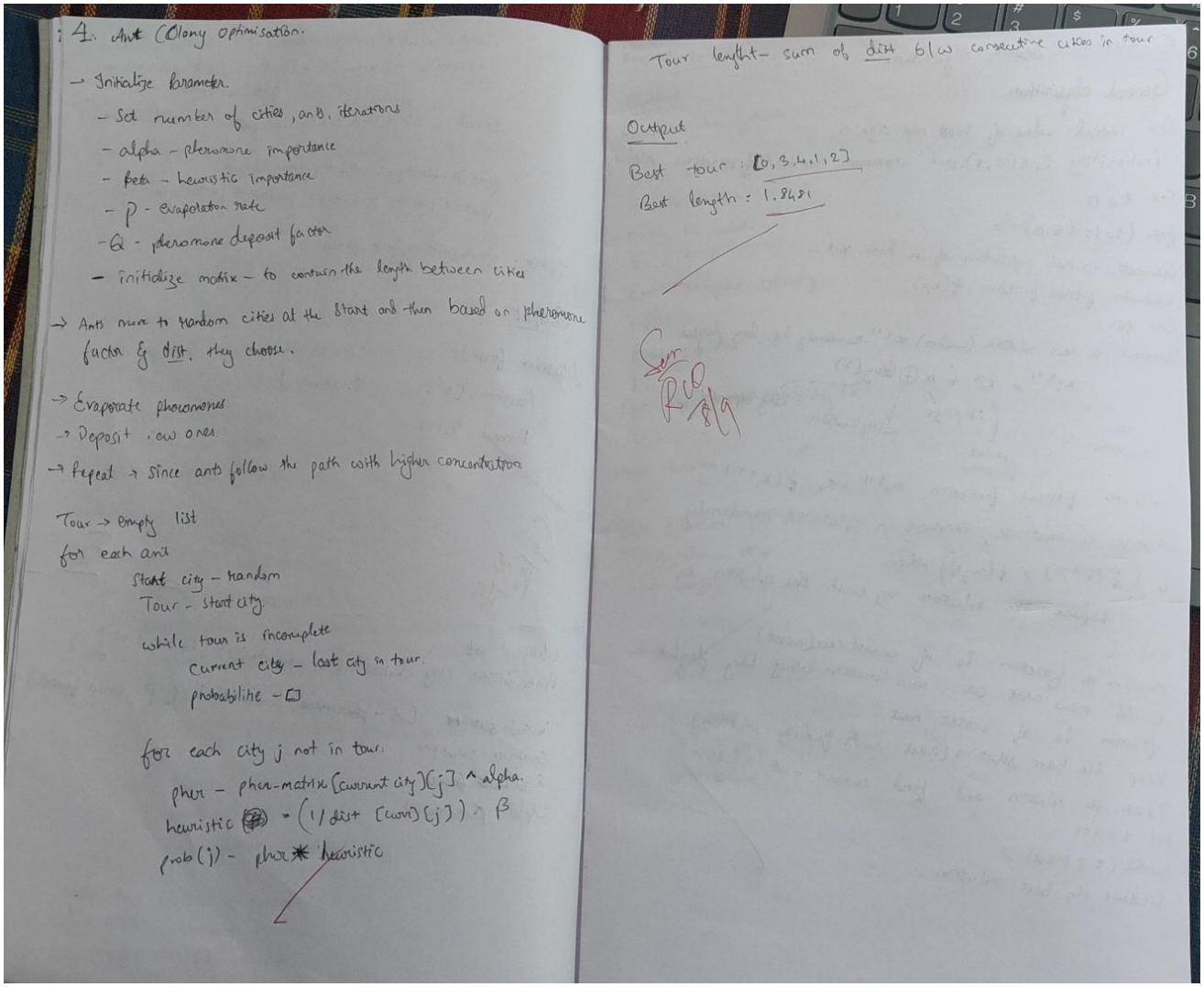
```
import random
import numpy as np

def fitness_function(position):
    x, y = position
    return -(x**2 + y**2 - 4*x - 6*y)

def particle_swarm_optimization(dimensions, num_particles, max_iterations, threshold):
    w = 0.5
    c1 = 1.2
    c2 = 1.4

    swarm = []
    for _ in range(num_particles):
        position = np.random.uniform(-10, 10, size=dimensions)
        velocity = np.random.uniform(-1, 1, size=dimensions)
        pbest_position = position.copy()
        pbest_fitness = fitness_function(position)
        swarm.append({'position': position, 'velocity': velocity,
                      'pbest_position': pbest_position, 'pbest_fitness': pbest_fitness})

    gbest_position = np.zeros(dimensions)
    gbest_fitness = -float('inf')

    for i in range(max_iterations):
        for p in swarm:
            fitness = fitness_function(p['position'])

            if fitness > p['pbest_fitness']:
                p['pbest_fitness'] = fitness
                p['pbest_position'] = p['position'].copy()

            if fitness > gbest_fitness:
                gbest_fitness = fitness
                gbest_position = p['position'].copy()

        if gbest_fitness >= threshold:
            print(f"Early stopping at iteration {i}")
            break

        for p in swarm:
            rand1 = random.random()
            rand2 = random.random()

            inertia = w * p['velocity']
            cognitive = c1 * rand1 * (p['pbest_position'] - p['position'])
            social = c2 * rand2 * (gbest_position - p['position'])

            p['velocity'] = inertia + cognitive + social
```

```
            p['position'] = p['position'] + p['velocity']

    print("SOLUTION FOUND:")
    print(f"  Position: {gbest_position}")
    print(f"  Fitness: {gbest_fitness}")
    return gbest_position, gbest_fitness

particle_swarm_optimization(dimensions=2, num_particles=20, max_iterations=5000, threshold=2)
```

Program4: Ant Colony Optimization for the Traveling Salesman Problem:

The foraging behaviour 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:



Code:

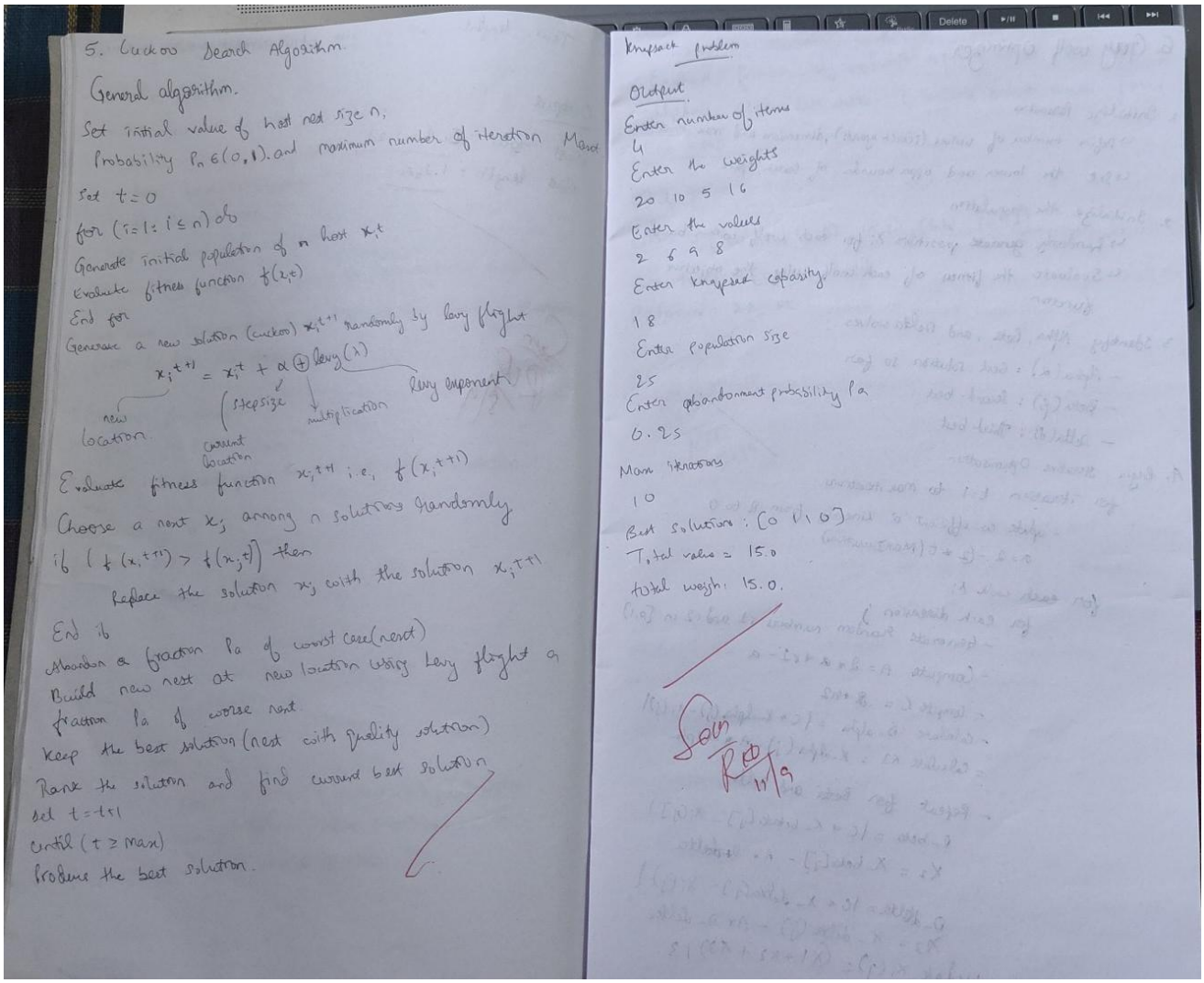
```
import numpy as np
import random


NUM_CITIES = 5
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
RHO = 0.5
Q = 100


cities = np.random.rand(NUM_CITIES, 2)
distance_matrix = np.linalg.norm(cities[:, None] - cities, axis=2)
pheromone_matrix = np.ones((NUM_CITIES, NUM_CITIES))


def calculate_probabilities(current_city, visited):
    probabilities = []
    for next_city in range(NUM_CITIES):
        if next_city in visited:
            probabilities.append(0)
        else:
            pheromone = pheromone_matrix[current_city][next_city] ** ALPHA
            heuristic = (1 / distance_matrix[current_city][next_city]) ** BETA
            probabilities.append(pheromone * heuristic)
    total = sum(probabilities)
    return [p / total if total > 0 else 0 for p in probabilities]

def construct_tour():
    start_city = random.randint(0, NUM_CITIES - 1)
    tour = [start_city]
    while len(tour) < NUM_CITIES:
        probs = calculate_probabilities(tour[-1], tour)
        next_city = np.random.choice(range(NUM_CITIES), p=probs)
        tour.append(next_city)
    return tour

def compute_tour_length(tour):
    return sum(distance_matrix[tour[i]][tour[(i + 1) % NUM_CITIES]] for i in range(NUM_CITIES))


best_tour = None
best_length = float('inf')

for iteration in range(NUM_ITERATIONS):
    all_tours = []

    for _ in range(NUM_ANTS):
```

```
        tour = construct_tour()
        length = compute_tour_length(tour)
        all_tours.append((tour, length))

        if length < best_length:
            best_tour = tour
            best_length = length


    pheromone_matrix *= (1 - RHO)


    for tour, length in all_tours:
        for i in range(NUM_CITIES):
            a, b = tour[i], tour[(i + 1) % NUM_CITIES]
            pheromone_matrix[a][b] += Q / length
            pheromone_matrix[b][a] += Q / length  # symmetric TSP


clean_tour = [int(city) for city in best_tour]
print("Best tour:", clean_tour)

print("Best length:", round(best_length, 4))
```

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.

Algorithm:



Code:

```
import numpy as np
import math

# --- Levy flight ---
def levy_flight(Lambda, dim):
    sigma = (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, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = u / abs(v)**(1 / Lambda)
    return step

# --- Sigmoid for binary conversion ---
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# --- Fitness function for knapsack ---
def fitness_function(x_bin, weights, values, capacity):
    total_weight = np.sum(x_bin * weights)
    total_value = np.sum(x_bin * values)
    if total_weight > capacity:
        return -1  # Penalize overweight solutions heavily
    else:
        return total_value

# --- Cuckoo Search for Binary Knapsack ---
def cuckoo_search_knapsack(weights, values, capacity, n=25, Pa=0.25, Maxt=500):
    dim = len(weights)
    # Initialize nests (continuous vectors)
    nests = np.random.uniform(low=-1, high=1, size=(n, dim))
    # Convert to binary solutions
    nests_bin = np.array([sigmoid(nest) > np.random.rand(dim) for nest in nests])
    fitness = np.array([fitness_function(x, weights, values, capacity) for x in nests_bin])

    best_idx = np.argmax(fitness)
    best_nest = nests[best_idx].copy()
    best_bin = nests_bin[best_idx].copy()
    best_fitness = fitness[best_idx]

    t = 0
    while t < Maxt:
        for i in range(n):
            # Generate new solution by Levy flight
            step = levy_flight(1.5, dim)
            new_nest = nests[i] + 0.01 * step
            # Convert new_nest to binary
            new_bin = sigmoid(new_nest) > np.random.rand(dim)
            new_fitness = fitness_function(new_bin, weights, values, capacity)

            # If new solution is better, replace
```

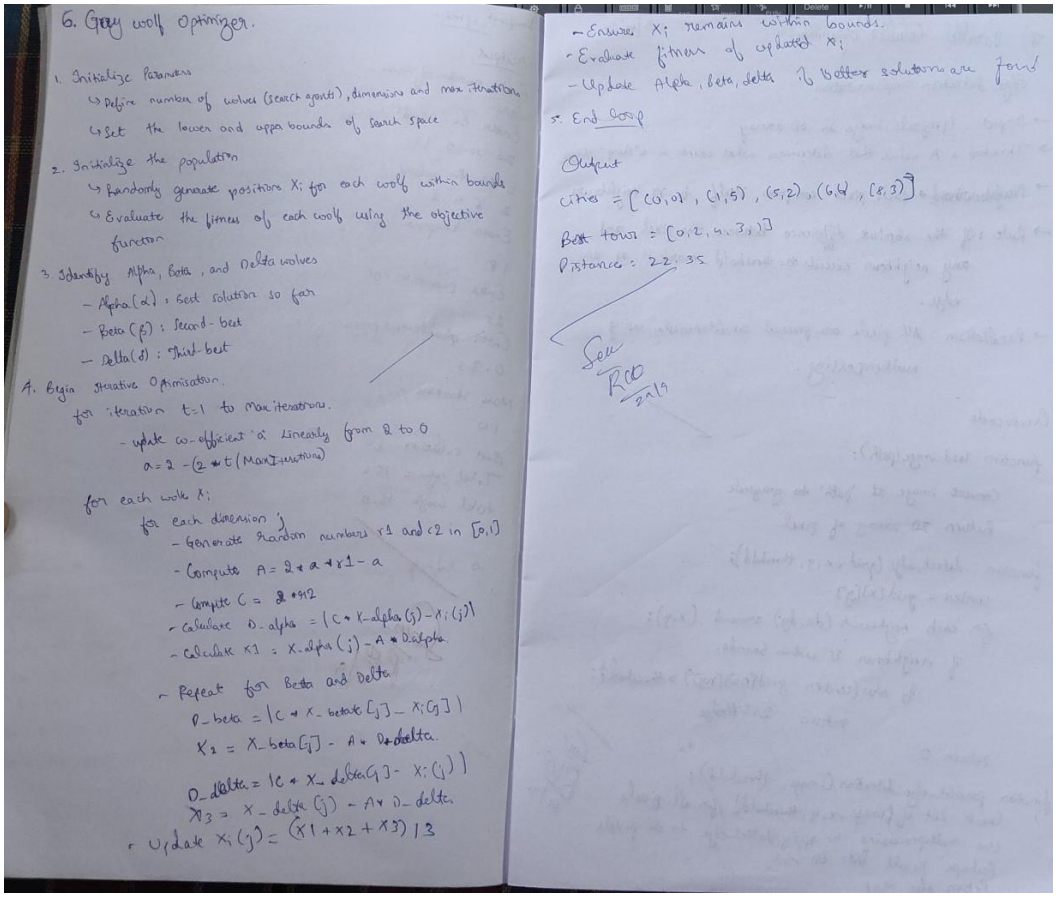
```
            if new_fitness > fitness[i]:
                nests[i] = new_nest
                nests_bin[i] = new_bin
                fitness[i] = new_fitness

                if new_fitness > best_fitness:
                    best_fitness = new_fitness
                    best_nest = new_nest.copy()
                    best_bin = new_bin.copy()

        # Abandon fraction Pa of worst nests
        num_abandon = int(Pa * n)
        worst_indices = np.argsort(fitness)[:num_abandon]
        for idx in worst_indices:
            nests[idx] = np.random.uniform(-1, 1, dim)
            nests_bin[idx] = sigmoid(nests[idx]) > np.random.rand(dim)
            fitness[idx] = fitness_function(nests_bin[idx], weights, values, capacity)

            if fitness[idx] > best_fitness:
                best_fitness = fitness[idx]
                best_nest = nests[idx].copy()
                best_bin = nests_bin[idx].copy()

        t += 1

    return best_bin, best_fitness

if __name__ == "__main__":
    print("Enter the number of items:")
    n_items = int(input())

    weights = []
    values = []

    print("Enter the weights of the items (space-separated):")
    weights = np.array(list(map(float, input().split())))
    if len(weights) != n_items:
        raise ValueError("Number of weights does not match number of items.")

    print("Enter the values of the items (space-separated):")
    values = np.array(list(map(float, input().split())))
    if len(values) != n_items:
        raise ValueError("Number of values does not match number of items.")

    print("Enter the knapsack capacity:")
    capacity = float(input())

    print("Enter population size (default 25):")
    n = input()
    n = int(n) if n.strip() else 25

    print("Enter abandonment probability Pa (default 0.25):")
```

```
    Pa = input()
    Pa = float(Pa) if Pa.strip() else 0.25

    print("Enter maximum iterations Maxt (default 500):")
    Maxt = input()
    Maxt = int(Maxt) if Maxt.strip() else 500

    best_solution, best_value = cuckoo_search_knapsack(weights, values, capacity, n=n, Pa=Pa,
Maxt=Maxt)

    print("\nBest solution (items selected):", best_solution.astype(int))
    print("Total value:", best_value)
    print("Total weight:", np.sum(best_solution * weights))
```

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.

Algorithm:



Code:

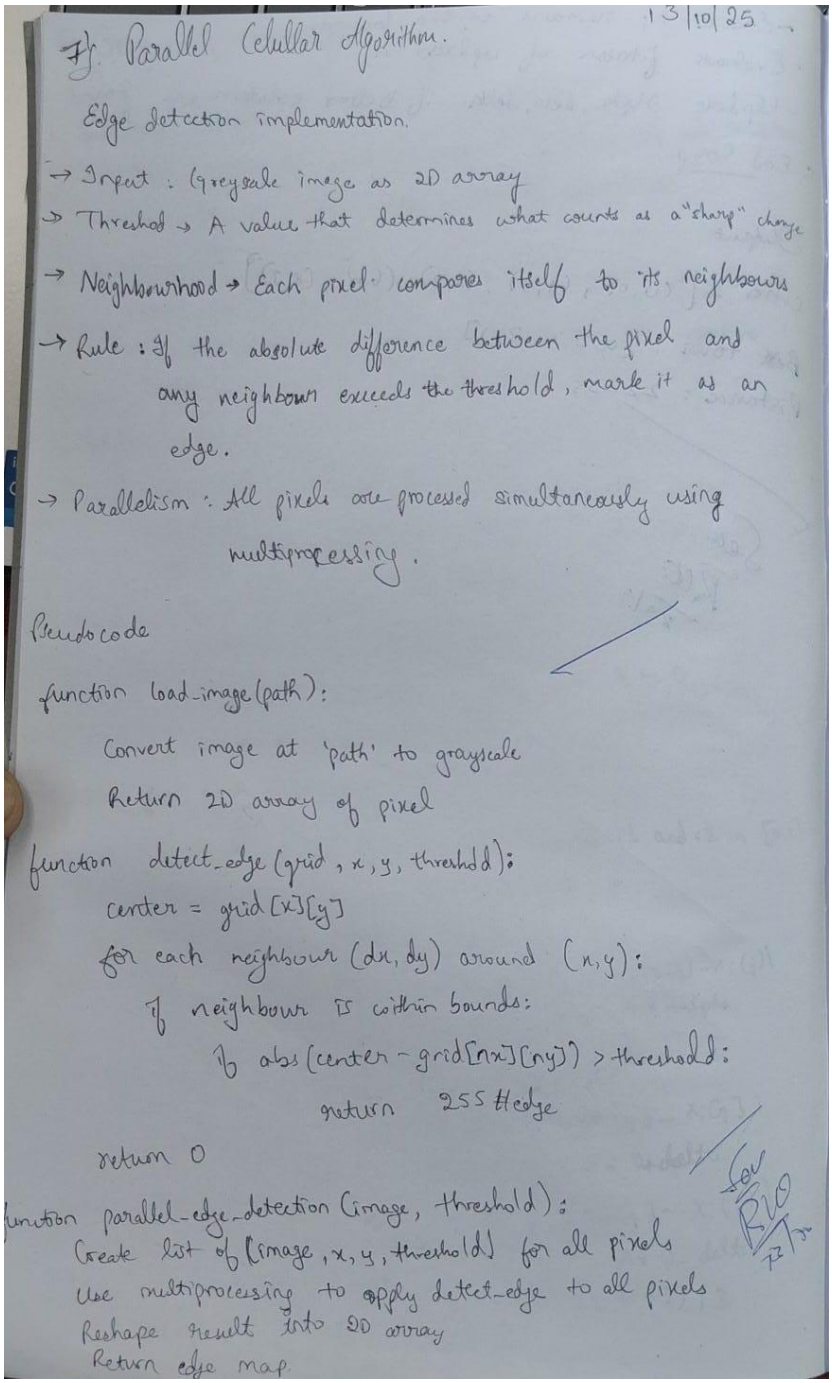
```
import numpy as np
import random

def distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
```

```
        for j in range(n):
            dist[i][j] = np.linalg.norm(np.array(cities[i]) - np.array(cities[j]))
    return dist

def tour_length(tour, dist):
    return sum(dist[tour[i]][tour[(i+1)%len(tour)]] for i in range(len(tour)))

def initialize_population(num_wolves, num_cities):
    return [random.sample(range(num_cities), num_cities) for _ in range(num_wolves)]

def gwo_tsp(cities, num_wolves=20, max_iter=100):
    dist = distance_matrix(cities)
    population = initialize_population(num_wolves, len(cities))
    fitness = [tour_length(tour, dist) for tour in population]

    alpha, beta, delta = sorted(zip(population, fitness), key=lambda x: x[1])[:3]

    for iter in range(max_iter):
        a = 2 - iter * (2 / max_iter)
        new_population = []

        for wolf in population:
            new_tour = []
            for i in range(len(cities)):
                r1, r2 = random.random(), random.random()
                A1 = 2 * a * r1 - a
                C1 = 2 * r2
                D_alpha = abs(C1 * alpha[0][i] - wolf[i])
                X1 = alpha[0][i] - A1 * D_alpha

                # Repeat for beta and delta
                # Combine X1, X2, X3 and discretize
                new_tour.append(int(X1) % len(cities))

            # Ensure it's a valid permutation
            new_tour = list(dict.fromkeys(new_tour))
            while len(new_tour) < len(cities):
                new_tour.append(random.choice([i for i in range(len(cities)) if i not in new_tour]))

            new_population.append(new_tour)

        population = new_population
        fitness = [tour_length(tour, dist) for tour in population]
        alpha, beta, delta = sorted(zip(population, fitness), key=lambda x: x[1])[:3]

    return alpha[0], alpha[1]

# Example usage
cities = [(0,0), (1,5), (5,2), (6,6), (8,3)]
best_tour, best_distance = gwo_tsp(cities)
print("Best tour:", best_tour)
print("Distance:", best_distance)
```

Program 7: 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:



7. Parallel Cellular Algorithm. 13/10/25

Edge detection implementation.

→ Input : Greyscale image as 2D array
→ Threshold → A value that determines what counts as a "sharp" change
→ Neighbourhood → Each pixel compares itself to its neighbours
→ Rule : If the absolute difference between the pixel and any neighbour exceeds the threshold, mark it as an edge.
→ Parallelism : All pixels are processed simultaneously using multiprocessing.

Pseudocode

```
function load_image(path):
    Convert image at 'path' to grayscale
    Return 2D array of pixel

function detect_edge(grid, x, y, threshold):
    center = grid[x][y]
    for each neighbour (dx, dy) around (x,y):
        if neighbour is within bounds:
            if abs(center - grid[nx][ny]) > threshold:
                return 255 #edge
    return 0

function parallel_edge_detection(image, threshold):
    Create list of [image, x, y, threshold] for all pixels
    Use multiprocessing to apply detect_edge to all pixels
    Reshape result into 2D array
    Return edge map
```

Code:

```
import numpy as np
from multiprocessing import Pool
from PIL import Image

# Load image and convert to grayscale
def load_image(path):
    img = Image.open(path).convert('L')  # 'L' mode = grayscale
    return np.array(img)

# Edge detection rule for a single pixel
def detect_edge(args):
    grid, x, y, threshold = args
    rows, cols = grid.shape
    center = grid[x][y]
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x + dx, y + dy
            if 0 <= nx < rows and 0 <= ny < cols:
                if abs(int(center) - int(grid[nx][ny])) > threshold:
                    return 255  # Edge
    return 0  # Non-edge

# Parallel cellular edge detection
def parallel_edge_detection(image, threshold=20):
    rows, cols = image.shape
    args = [(image, x, y, threshold) for x in range(rows) for y in range(cols)]
    with Pool() as pool:
        edges = pool.map(detect_edge, args)
    return np.array(edges).reshape((rows, cols))

# Save or display result
def save_edge_image(edge_array, output_path='edges.png'):
    edge_img = Image.fromarray(edge_array.astype(np.uint8))
    edge_img.save(output_path)
    edge_img.show()

# Example usage
if __name__ == '__main__':
    image = load_image('your_image.jpg')  # Replace with actual image path
    edges = parallel_edge_detection(image, threshold=30)
    save_edge_image(edges)
```