VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

PADMASHREE JAIN D(1BM23CS223)

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 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 **PADMASHREE JAIN D(1BM23CS223) ,** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

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

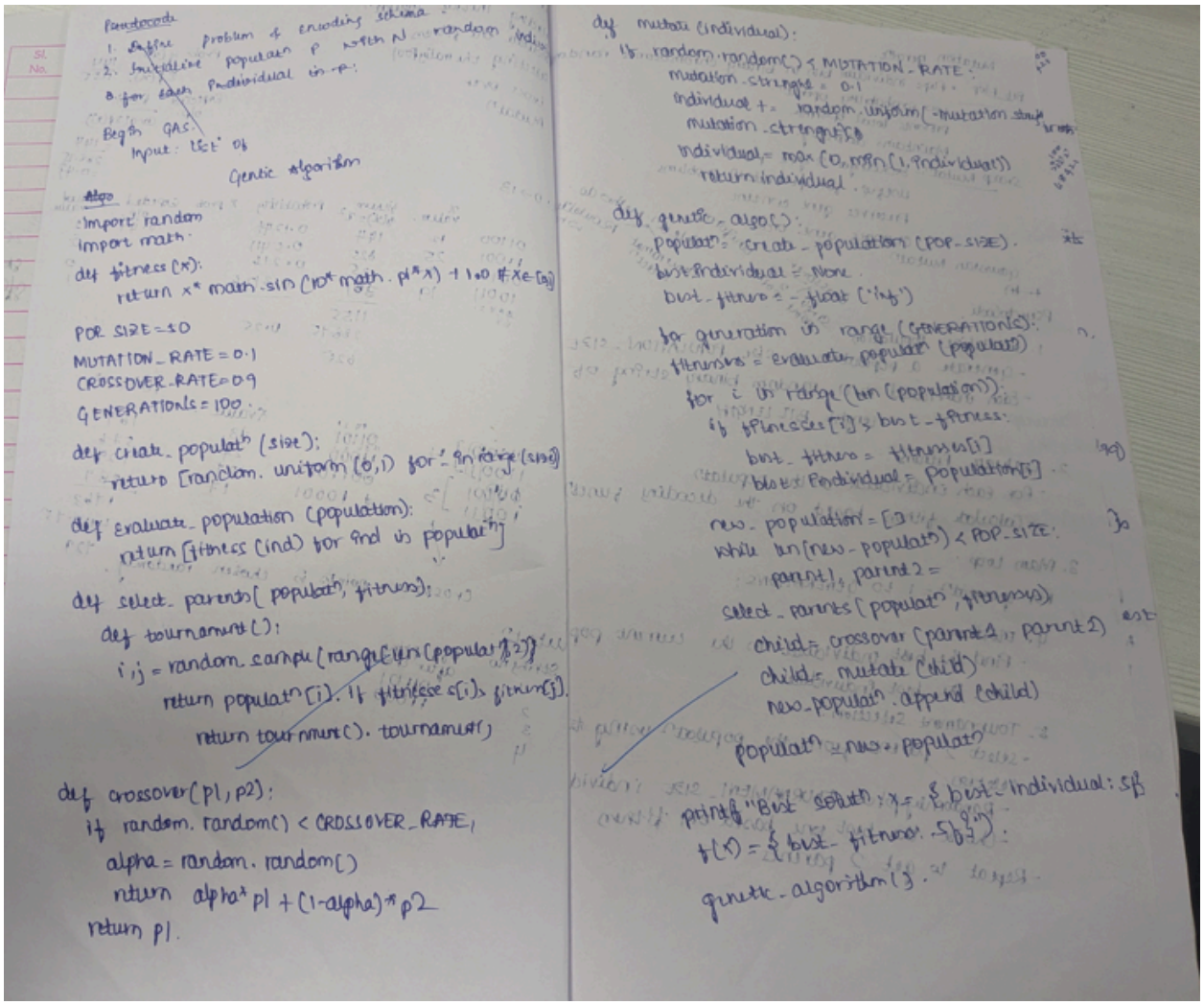
Index

2	25/8/25	Gene Expression Algorithm	5
3	1/9/25	Particle Swarm Optimisation	8
4	8/9/25	Ant Colony Optimisation	11
5	15/9/25	Cuckoo Search Optimisation	15
6	29/9/25	Grey Wolf Optimisation	19
7	13/10/25	Parallel Cellular Algorithm	22

Github Link:
https://github.com/PADMASHREE719/bio-inspired-system

Program 1

Design and implement a **genetic algorithm (Knapsack)**in Python



Pseudocode
1. Define problem & encoding scheme
2. Initialize population P with N random individuals
3. for each individual in P:

Begin GAS:
Input: List of

Genetic Algorithm

Algo
import random
import math
def fitness(x):
return x* math.sin(10* math.pi*x) + 1.0 # x∈[0,2]

POP_SIZE=50
MUTATION_RATE=0.1
CROSSOVER_RATE=0.9
GENERATIONS=100

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

def evaluate_population(population):
return [fitness(ind) for ind in population]

def select_parents(population, fitness):
def tournament():
i,j = random.sample(range(len(population)),2)
return population[i] if fitness[i]>fitness[j]
return tournament(), tournament()

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

def mutate(individual):
if random.random() < MUTATION_RATE:
mutation_strength = 0.1
individual += random.uniform(-mutation_strength,
mutation_strength)
individual = max(0, min(2, individual))
return individual

def genetic_algo():
population = create_population(POP_SIZE)
best_individual = None
best_fitness = -float('inf')

for generation in range(GENERATIONS):
fitness = evaluate_population(population)
for i in range(len(population)):
if fitness[i] > best_fitness:
best_fitness = fitness[i]
best_individual = population[i]
new_population = []
while len(new_population) < POP_SIZE:
parent1, parent2 =
select_parents(population, fitness)
child = crossover(parent1, parent2)
child = mutate(child)
new_population.append(child)
population = new_population

print("Best solution: x = {best_individual:.5f}
f(x) = {best_fitness:.5f}")

genetic_algorithm()

Mutation

Bit Flip - flips individual bits in binary chromosomes randomly
- low probability operator
- Prevents local optima
- Maintains diversity

Swap Mutation - exchanges 2 gene positions
- useful for permutation problems
- Preserves gene content
- Changes gene order

Gaussian Mutation

Crossover
- Single point
- 2 point
- Uniform

Selection Methods
- Tournament selection
- Roulette wheel

Pseudocode

1. Initialize Population
- Generate a population of size POPULATION_SIZE
- Each individual is a random binary string of length BIT_LENGTH

2. Evaluate fitness
- For each individual in the population
- Calculate fitness based on the decoding function

3. Main Loop
For generation = 1 to GENERATIONS:

4. Elitism
- Find the best individual in the current population
- Save the best individual

5. Tournament Selection
- Select 2 parents from the population using tournament selection
- Randomly pick TOURNAMENT_SIZE individuals
- Choose the best one based on fitness
- Repeat to get 2 parents

6. Crossover
- Perform crossover between parents.
- Randomly select a crossover point
- Create 2 children by exchanging parts of the parents chromosomes at the crossover point

7. Mutation
- For each child generated:
- With probability MUTATION_RATE, flip random bit in child's chromosome.

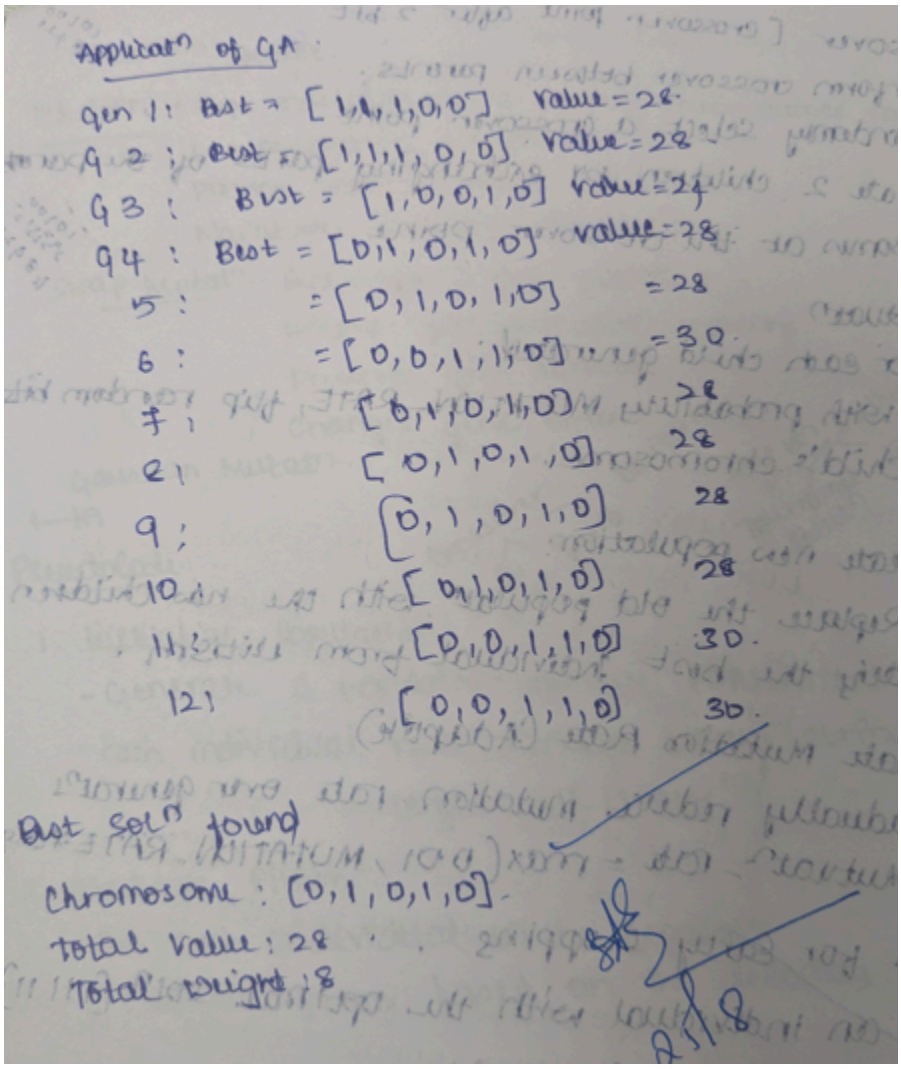
8. Create new population.
- Replace the old population with the new children, including the best individual from elitism.

9. Update Mutation Rate (Adaptive)
- Gradually reduce mutation rate over generations
- Mutation_rate = max(0.01, MUTATION_RATE * 0.99)

10. Check for early stopping.
- If an individual with the optimal solution (1111) is found, stop the algo early.

11. Return Best Solution.
After all generations or early stopping, return the best individual found

Ex



Code:

```
import random

# --------------------------------
# Problem Setup
# --------------------------------
items = [
    (2, 6),   # (weight, value)
    (3, 10),
    (4, 12),
    (5, 18),
    (9, 22)
]
capacity = 10
POP_SIZE = 6      # Number of chromosomes
GENS = 15         # Number of generations
MUT_RATE = 0.1    # Mutation probability

# --------------------------------
```

```
# Helper Functions
# ------------------------------

def fitness(chromosome):
    weight, value = 0, 0
    for gene, (w, v) in zip(chromosome, items):
        if gene == 1:
            weight += w
            value += v
    if weight > capacity:
        return 0  # Invalid solution
    return value

def create_chromosome():
    return [random.randint(0, 1) for _ in range(len(items))]

def selection(pop, fits):
    # Roulette Wheel Selection
    total_fit = sum(fits)
    if total_fit == 0:
        return random.choice(pop)
    pick = random.uniform(0, total_fit)
    current = 0
    for chrom, fit in zip(pop, fits):
        current += fit
        if current >= pick:
            return chrom

def crossover(p1, p2):
    point = random.randint(1, len(items)-1)
    return p1[:point] + p2[point:], p2[:point] + p1[point:]

def mutate(chrom):
    for i in range(len(chrom)):
        if random.random() < MUT_RATE:
            chrom[i] = 1 - chrom[i]  # Flip bit
    return chrom

# ------------------------------
# Genetic Algorithm
# ------------------------------

population = [create_chromosome() for _ in range(POP_SIZE)]

for gen in range(GENS):
    fitnesses = [fitness(c) for c in population]
    best_fit = max(fitnesses)
    best_chrom = population[fitnesses.index(best_fit)]
```

```
    print(f"Generation {gen+1}: Best={best_chrom} Value={best_fit}")

    new_population = []
    while len(new_population) < POP_SIZE:
        parent1 = selection(population, fitnesses)
        parent2 = selection(population, fitnesses)
        child1, child2 = crossover(parent1, parent2)
        new_population.append(mutate(child1))
        if len(new_population) < POP_SIZE:
            new_population.append(mutate(child2))
    population = new_population

# Final Result
fitnesses = [fitness(c) for c in population]
best_fit = max(fitnesses)
best_chrom = population[fitnesses.index(best_fit)]

print("\nBest Solution Found:")
print(f"Chromosome: {best_chrom}")
print(f"Total Value: {best_fit}")
print(f"Total Weight: {sum(w for gene,(w,v) in zip(best_chrom,items) if gene)}")
```


Program 2

gene expression algorithm for finding Travelling salesman problem

Algorithm:

Pseudocode for GEA applied to Routing (TSP):

Algorithm GEA_TSP

1. Define the problem:
   - Input : set of cities with distances.
   - Output: shortest possible route visiting all c

2. Initialize parameters:
   popul^n_size, mutat^n_rate, crossover_rate, gene

3. Initialise popul^n.
   Create random routes (permutat? of cities)

4. Evaluate fitness:
   for each route in population:
      fitness = 1 / (total_distance(route))
5. Repeat until max generations
   a. Selection:
      - select parent routes based on fitness
   b. Crossover:
      - Perform crossover b/n parents to create children
      - Example: Order Crossover (Ox)
   c. Mutation:
      - Randomly swap cities in the route with small probability.
   d. Gene Expression:
      - Ensure valid route (all cities appear exactly once)
   e. Evaluate new population's fitness.
   f. Replace old population with new population
6. Output best solution (shortest route found)

Ip: Best distance: 20 15
Best Route: [0,1,3,2,0] [0,1,2,3,0].

Simple example:

Distance Matrix:

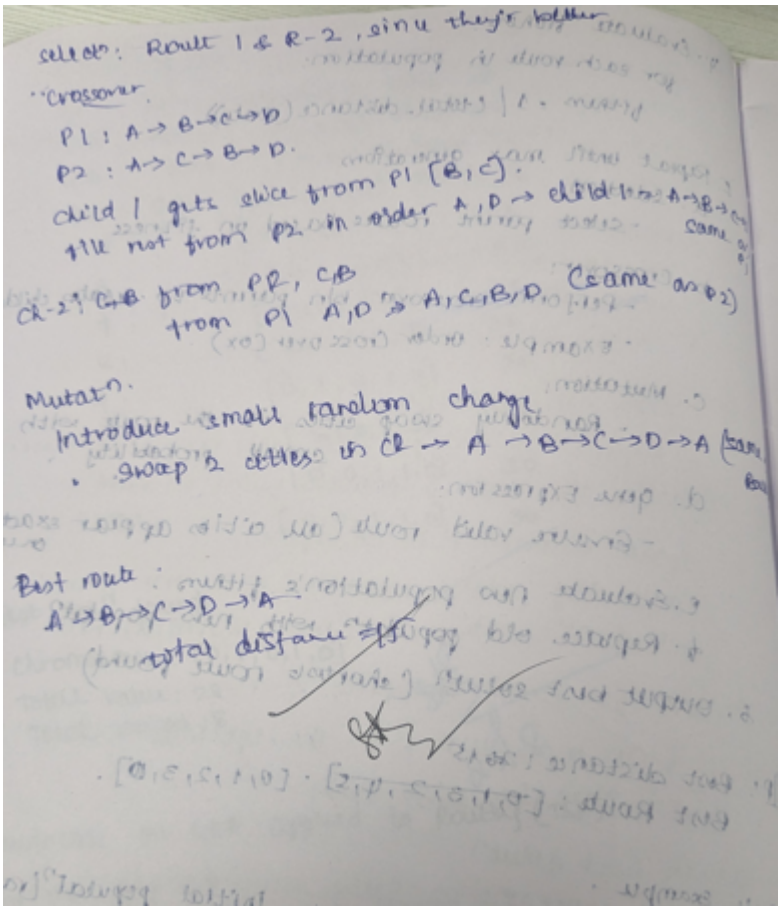
	A	B	C	D
A	0	4	1	3
B	1	0	2	5
C	1	2	0	6
D	3	5	6	0

S-1: Initial population (random Routes):
A → B → C → D.
A → C → B → D
A → D → B → C
A → C → D → B

Evaluate fitness (calculate the round-trip distance).

A → B → C → D → A = 15
4 2 6 3
A → C → B → D → A = 17

Shortest = 15



Code:

```
import random

# ---- Step 1: Problem Setup ----
cities = list(range(6))  # 6 cities labeled 0 to 5
distance_matrix = [
    [0, 2, 9, 10, 7, 3],
    [2, 0, 6, 4, 3, 8],
    [9, 6, 0, 8, 5, 6],
    [10, 4, 8, 0, 6, 7],
    [7, 3, 5, 6, 0, 4],
    [3, 8, 6, 7, 4, 0]
]

# Parameters
POP_SIZE = 20
MUTATION_RATE = 0.1
GENERATIONS = 200

# ---- Step 2: Fitness Function ----
def total_distance(route):
    distance = 0
    for i in range(len(route)):
        distance += distance_matrix[route[i]][route[(i+1) % len(route)]]
```

```
    return distance

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

# ---- Step 3: Initialize Population ----
def create_route():
    route = cities[:]
    random.shuffle(route)
    return route

def initial_population():
    return [create_route() for _ in range(POP_SIZE)]

# ---- Step 4: Selection ----
def selection(population):
    population.sort(key=total_distance)
    return population[:POP_SIZE//2]  # take best half

# ---- Step 5: Crossover ---- (Order Crossover)
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [None]*len(parent1)
    child[start:end] = parent1[start:end]

    fill_values = [gene for gene in parent2 if gene not in child]
    pos = 0
    for i in range(len(child)):
        if child[i] is None:
            child[i] = fill_values[pos]
            pos += 1
    return child

# ---- Step 6: Mutation ----
def mutate(route):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(len(route)), 2)
        route[i], route[j] = route[j], route[i]
    return route

# ---- Step 7: Main Loop ----
def gene_expression_algorithm():
    population = initial_population()
    best_route = None
    best_distance = float('inf')

    for gen in range(GENERATIONS):
        selected = selection(population)
```

```
        children = []
        while len(children) < POP_SIZE:
            p1, p2 = random.sample(selected, 2)
            child = crossover(p1, p2)
            child = mutate(child)
            children.append(child)
        population = children

        # Track best
        current_best = min(population, key=total_distance)
        current_dist = total_distance(current_best)
        if current_dist < best_distance:
            best_route = current_best
            best_distance = current_dist

        if gen % 20 == 0:  # log progress
            print(f"Gen {gen}: Best Distance = {best_distance}")

    return best_route, best_distance

# ---- Run ----
best_route, best_distance = gene_expression_algorithm()
print("\nBest Route:", best_route)
print("Best Distance:", best_distance)
```

Program 3

Find the maximum of the function f(x)=−x^2+y^2 using **Particle Swarm Optimization** by iteratively updating particle positions and velocities.

Algorithm:

Particle swarm optimization.

X → position
V → velocity
w → inertia
c_2 → social
c_1 → cognitive const
r_1, r_2 → random value

Eg:
Obj func $f(x,y) = x^2+y^2$.
$x \in [-10, 10]$.
Number of particles: N
Max number of iterations: max_iter
Parameters: inertia weight w, cognitive co-efficient c_1.
social coefficient c_2

op: Maximum value $f(x^*, y^*)$.

Steps:

1. Initialize the swarm

for each particle $i = 1, 2 \dots N$
Randomly initialize position $P_i = (x_i, y_i)$ within bounds
Initialize velocity $v_i = (0, 0)$
Set personal best position $pbest_i = P_i$
Evaluate fitness $f(P_i)$ & set as personal best value f_{pbest}.

2. Determine the global best particle

find particle with the ~~best~~ max value.
set global best position $g_{best} = pbest_{best}$.
set global best fitness $f_{gbest} = \max(f_{pbest_i})$.

3. for each iteration $t = 1$ to max_iter:

for each particle i:

(a) Update velocity.

$$v_i^{t+1} = w \cdot v_i^t + c_1 r_1 (pbest_i - p_i^t) + c_2 r_2 (g_{best} - p_i^t)$$

where $r_1, r_2 \sim$ Uniform $(0,1)$

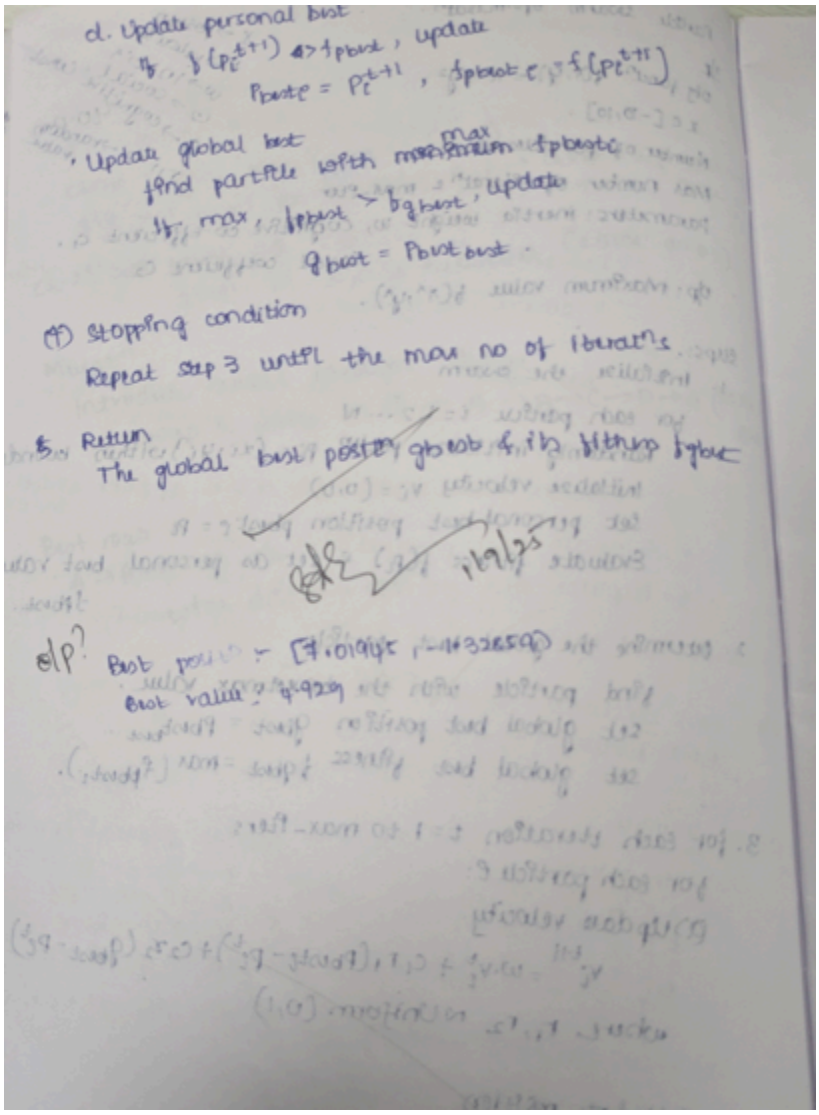
(b) Update position

$$P_i^{t+1} = P_i^t + v_i^{t+1}$$

Clip P_i^{t+1} to stay within bounds.

(c) Evaluate fitness

$$f(P_i^{t+1}) = (x_i^{t+1})^2 + (y_i^{t+1})^2$$



Code:

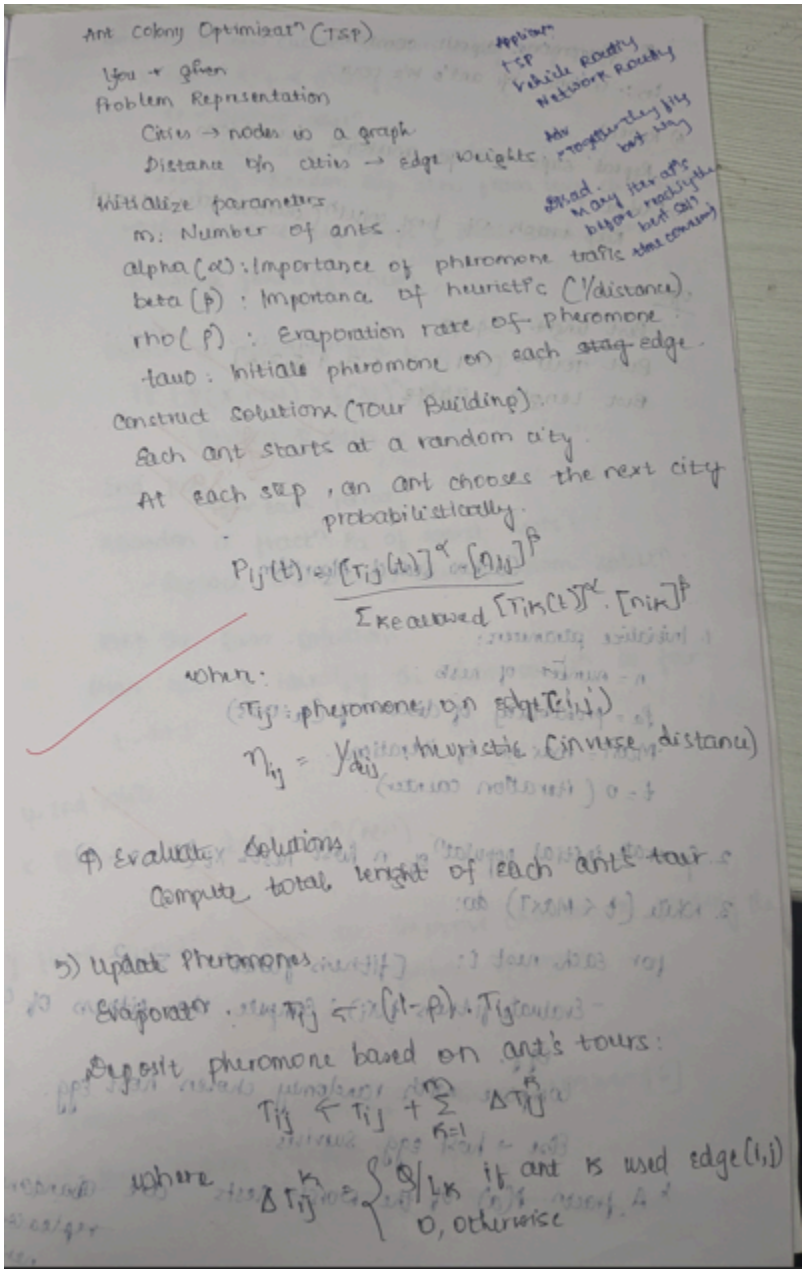
```
import numpy as np

# Objective function
def f(position):
    x, y = position
    return x**2 + y**2

class Particle:
    def __init__(self, bounds):
        self.position = np.array([np.random.uniform(bounds[0][0], bounds[0][1]),
                                  np.random.uniform(bounds[1][0], bounds[1][1])])
        self.velocity = np.zeros_like(self.position)
        self.pbest_position = self.position.copy()
        self.pbest_value = float('inf')

    def update_velocity(self, gbest_position, w, c1, c2):
        r1, r2 = np.random.rand(2)
```

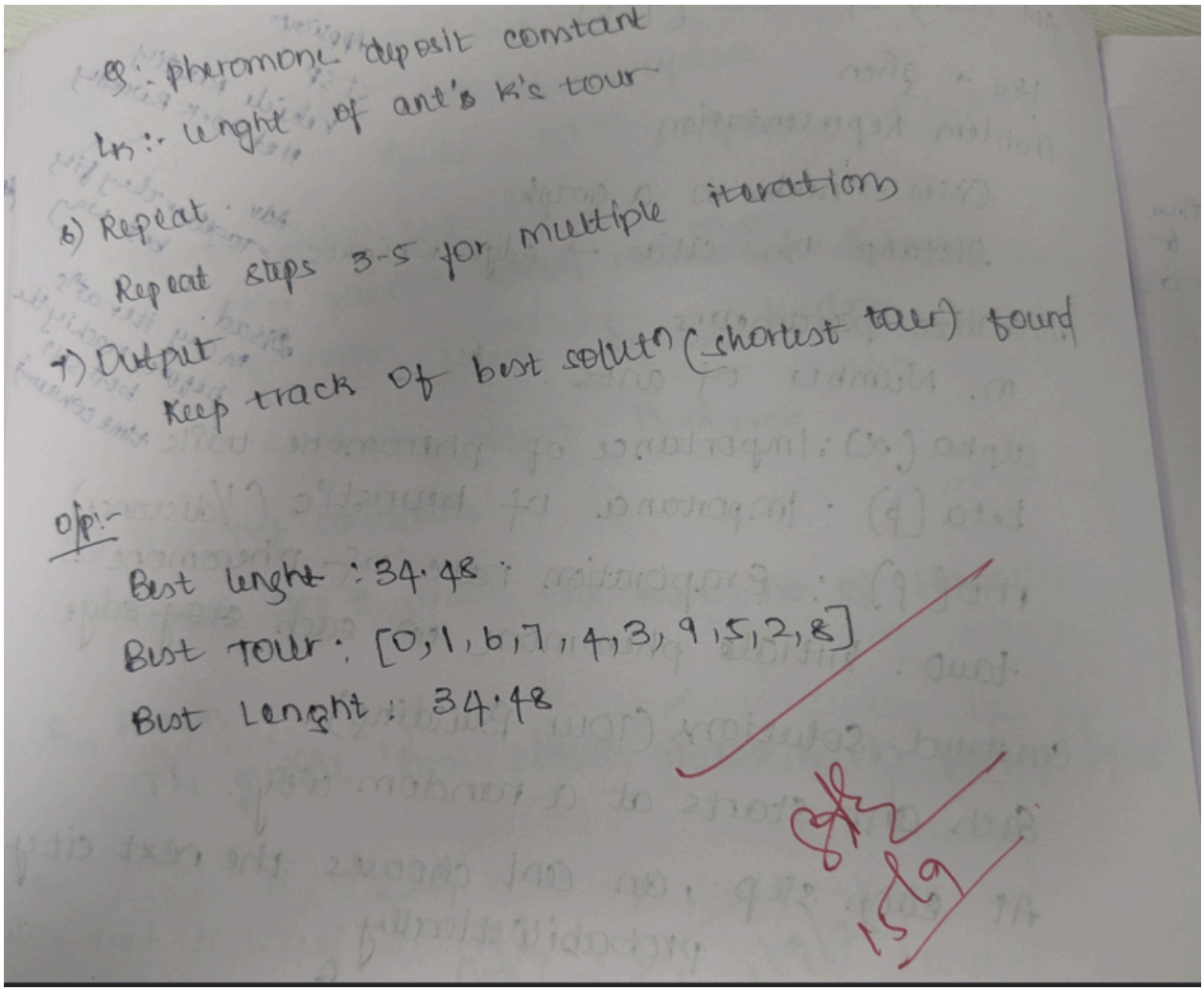
```
        cognitive = c1 * r1 * (self.pbest_position - self.position)
        social = c2 * r2 * (gbest_position - self.position)
        self.velocity = w * self.velocity + cognitive + social

    def update_position(self, bounds):
        self.position += self.velocity
        # Keep particle inside bounds
        for i in range(len(bounds)):
            self.position[i] = np.clip(self.position[i], bounds[i][0], bounds[i][1])

def pso(objective_function, bounds, num_particles, max_iter, w=0.5, c1=1.5,
c2=1.5):
    swarm = [Particle(bounds) for _ in range(num_particles)]
    gbest_value = float('inf')
    gbest_position = None

    for iteration in range(max_iter):
        for particle in swarm:
            fitness = objective_function(particle.position)

            # Update personal best
            if fitness < particle.pbest_value:
                particle.pbest_value = fitness
                particle.pbest_position = particle.position.copy()

            # Update global best
            if fitness < gbest_value:
                gbest_value = fitness
                gbest_position = particle.position.copy()

        # Update velocity and position for each particle
        for particle in swarm:
            particle.update_velocity(gbest_position, w, c1, c2)
            particle.update_position(bounds)

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

    return gbest_position, gbest_value

# Parameters
bounds = [(-10, 10), (-10, 10)]  # Search space for x and y
num_particles = 30
max_iter = 50

best_pos, best_val = pso(f, bounds, num_particles, max_iter)
print(f"Best position: {best_pos}, Best value: {best_val}")
```

Program 4

Find the shortest route visiting all cities once and returning to the start using **Ant Colony Optimization.**

Algorithm:





Code:

```
import numpy as np
import random
import math

# Step 1: Define the problem (set of cities with coordinates)
cities=[
 (0, 0),
 (1, 5),
 (5, 2),
 (6, 6),
 (8, 3),
 (7, 9),
 (2, 7),

 (3, 3)
]
N = len(cities)
```

```
# Calculate Euclidean distance matrix
def euclidean_distance(a, b):
 return math.sqrt((a[0] - b[0]) ** 2 + (a[1] - b[1]) ** 2)

distance_matrix = [[euclidean_distance(cities[i], cities[j]) for j in range(N)] for i in range(N)]

# Step 2: Initialize Parameters
num_ants = N
max_iterations = 100
alpha = 1.0
beta = 5.0
rho = 0.5
Q = 100.0
tau_0 = 1.0

pheromone = [[tau_0 for _ in range(N)] for _ in range(N)]
heuristic = [[1 / distance_matrix[i][j] if i != j else 0 for j in range(N)] for i in range(N)]

best_tour = None
best_length = float('inf')

# Step 3, 4, 5: Iterate
for iteration in range(max_iterations):
 all_tours = []
 all_lengths = []

 for ant in range(num_ants):
 unvisited = list(range(N))
 current_city = random.choice(unvisited)
 tour = [current_city]
 unvisited.remove(current_city)

 while unvisited:
 probabilities = []
 for j in unvisited:
 tau = pheromone[current_city][j] ** alpha
 eta = heuristic[current_city][j] ** beta
 probabilities.append(tau * eta)
 total = sum(probabilities)
 probabilities = [p / total for p in probabilities]
 next_city = random.choices(unvisited, weights=probabilities, k=1)[0]
tour.append(next_city)
 unvisited.remove(next_city)
 current_city = next_city
```



```
 # Return to starting city
 tour.append(tour[0])
```

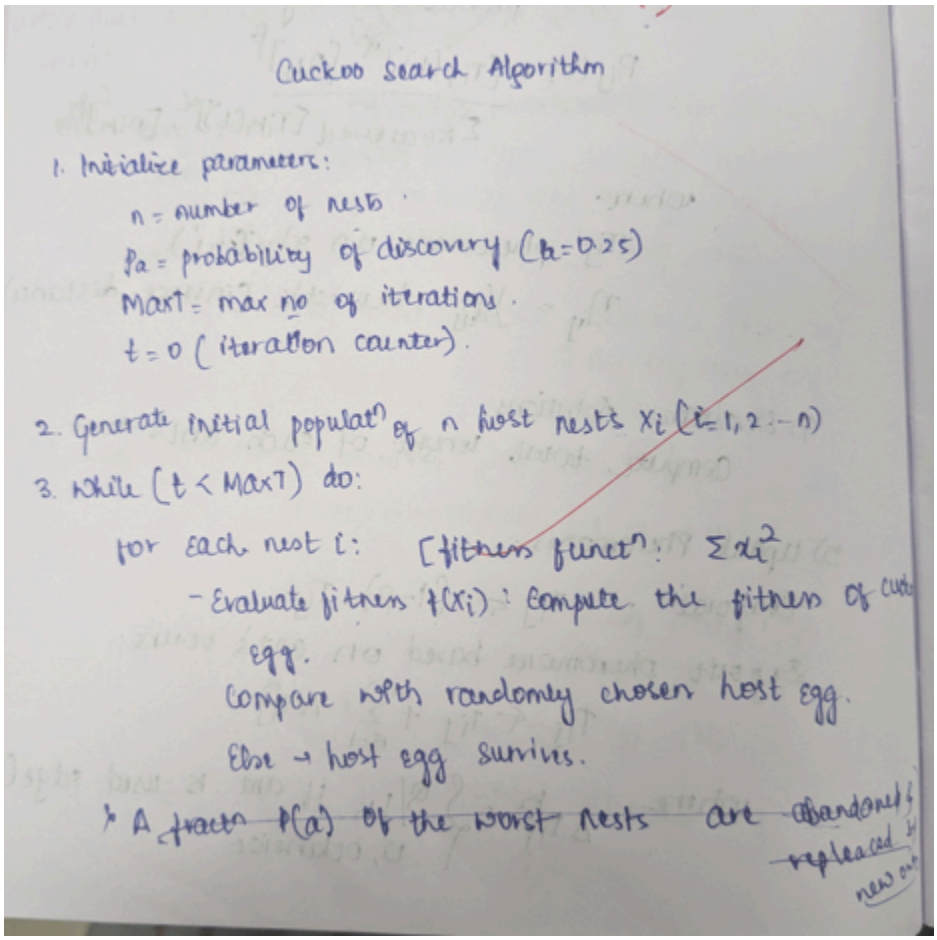
```
 tour_length = sum(distance_matrix[tour[i]][tour[i+1]] for i in range(N))
all_tours.append(tour)
 all_lengths.append(tour_length)

 if tour_length < best_length:
 best_length = tour_length
 best_tour = tour

 # Step 4: Update Pheromones
 # Evaporation
 for i in range(N):
 for j in range(N):
 pheromone[i][j] *= (1 - rho)

 # Deposit
 for tour, length in zip(all_tours, all_lengths):
 for i in range(N):
 a = tour[i]
 b = tour[i+1]
 pheromone[a][b] += Q / length
 pheromone[b][a] += Q / length # because TSP is symmetric

# Step 6: Output the Best Solution
print("Best tour:", best_tour)
print("Best tour length:", round(best_length, 2))
```

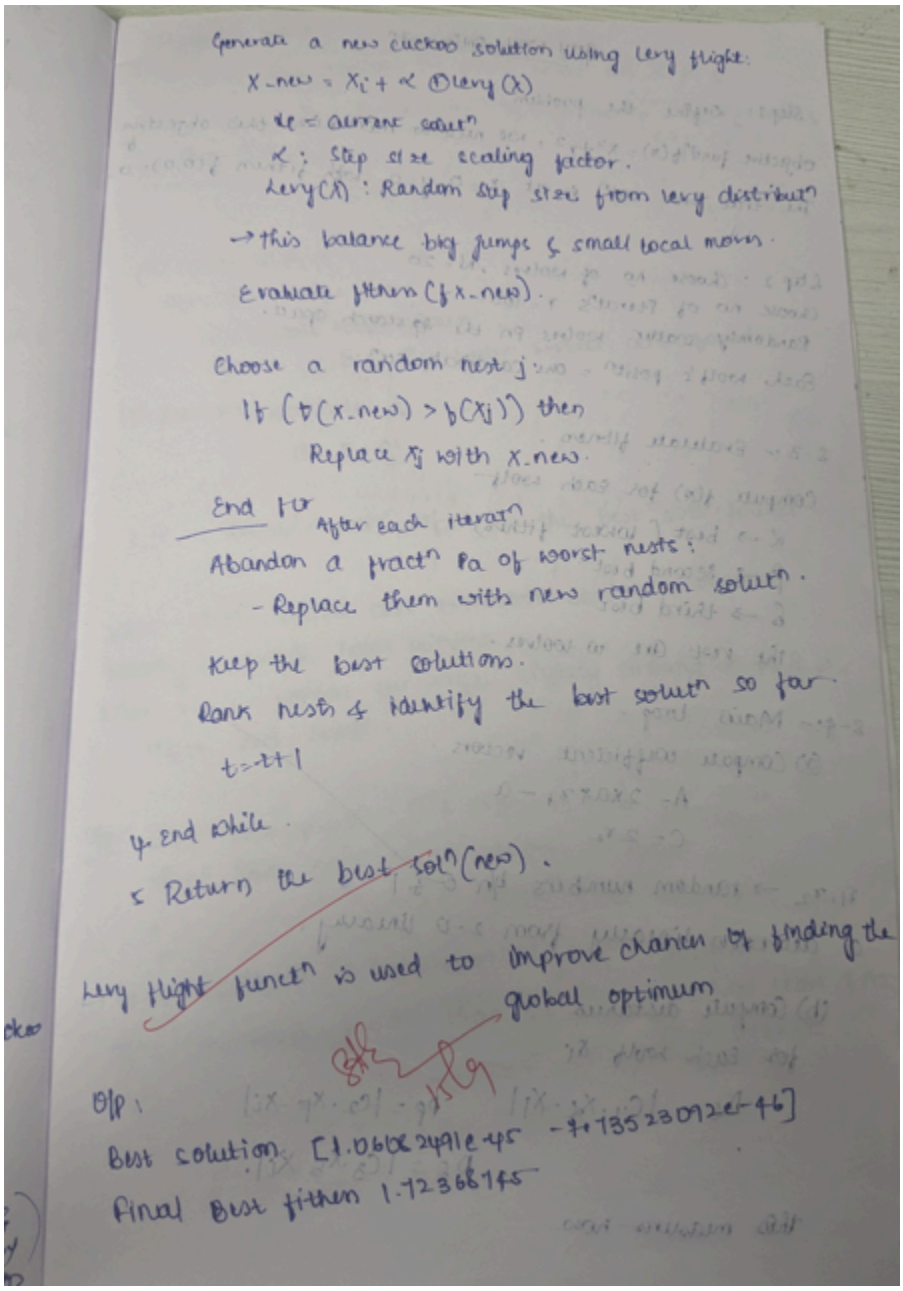

Program 5

Solve the 0/1 Knapsack Problem by using **Cuckoo Search** to maximize total item value

without exceeding the weight limit.

Algorithm:





Code:

```
import random
import numpy as np

# Step 1: Define the Knapsack Problem
# Sample data: values and weights of items
values = [60, 100, 120, 80, 30]
weights = [10, 20, 30, 15, 5]
max_weight = 50
num_items = len(values)

# Fitness function: maximize value while staying under weight limit
def fitness(nest):
 total_value = 0
```

```
 total_weight = 0
 for i in range(num_items):
 if nest[i] == 1:
 total_value += values[i]
```



```
 total_weight += weights[i]
 if total_weight > max_weight:
 return 0 # Penalty for exceeding capacity
 return total_value

# Step 2: Initialize Parameters
num_nests = 25
max_iterations = 100
discovery_rate = 0.25 # Probability of discovering a nest
alpha = 1.0 # Step size for Lévy flights

# Step 3: Initialize Population
def initialize_population():
 return [[random.randint(0, 1) for _ in range(num_items)] for _ in range(num_nests)]

population = initialize_population()

# Step 4: Evaluate Fitness
fitness_values = [fitness(nest) for nest in population]

# Step 5: Generate New Solutions using Lévy flights (simple binary flipping)
def levy_flight(nest):
 new_nest = nest.copy()
 for i in range(num_items):
 if random.random() < 0.3: # 30% chance to flip each bit
 new_nest[i] = 1 - new_nest[i]
 return new_nest

# Step 6: Replace worst nests
def abandon_worst_nests(population, fitness_values):
 num_abandon = int(discovery_rate * num_nests)
 worst_indices = np.argsort(fitness_values)[:num_abandon]

 for idx in worst_indices:
 population[idx] = [random.randint(0, 1) for _ in range(num_items)]
 return population

# Step 7: Main Loop
best_nest = None
best_fitness = -1

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

```
# Generate a new solution by Lévy flight
new_nest = levy_flight(population[i])
new_fitness = fitness(new_nest)

# Greedy selection
```

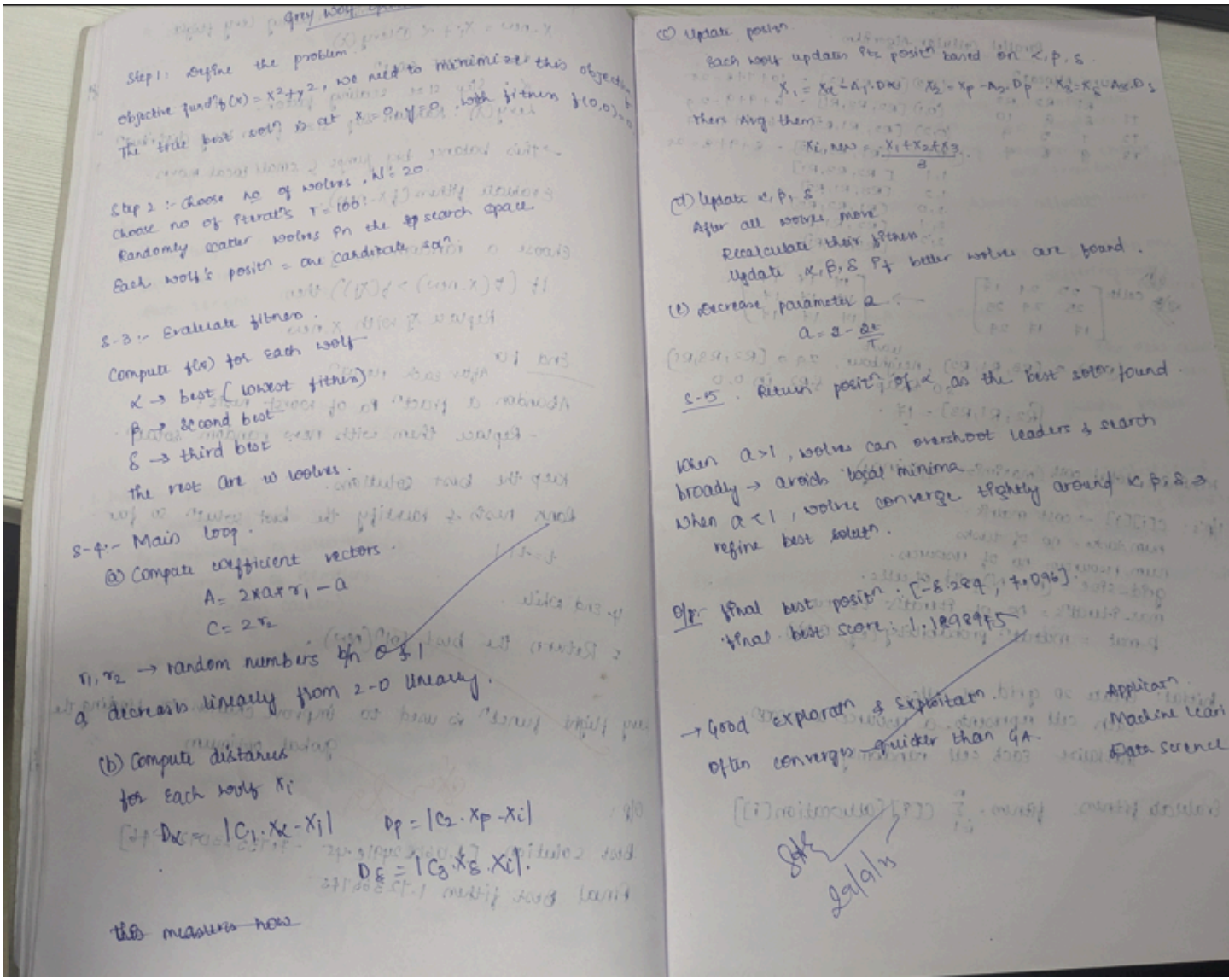


```
if new_fitness > fitness_values[i]:
population[i] = new_nest
fitness_values[i] = new_fitness

# Track the global best
if new_fitness > best_fitness:
best_fitness = new_fitness
best_nest = new_nest.copy()

# Abandon a fraction of the worst nests
population = abandon_worst_nests(population, fitness_values)
fitness_values = [fitness(nest) for nest in population]

# Step 8: Output the Best Solution
print("Best Solution (Item Inclusion):", best_nest)
print("Best Total Value:", best_fitness)
print("Total Weight:",
 sum(weights[i] for i in range(num_items) if best_nest[i] == 1))
```

Program 6

Optimize the assignment of cars to parking slots to minimize total walking distance using **Grey Wolf Optimization**, simulating the social hierarchy and hunting behavior of grey wolves to iteratively improve solutions.

Algorithm:



Code:

```
import numpy as np

# Example Data:
num_cars = 5
num_slots = 5
slot_distances = np.array([10, 20, 15, 30, 25]) # Distances of parking slots to entrance
num_wolves = 10
max_iter = 50

# Initialize wolves: each wolf is a permutation of slot indices representing car assignments
def initialize_population(num_wolves, num_slots):
```



```
 population = []
 for _ in range(num_wolves):
 wolf = np.random.permutation(num_slots)
```

```
 population.append(wolf)
 return population

# Fitness function: total walking distance for assigned slots
def fitness(wolf):
 total_distance = 0
 for car_index, slot_index in enumerate(wolf):
 total_distance += slot_distances[slot_index] # Distance of car's assigned slot
return total_distance

# Update position of wolf based on alpha, beta, delta (GWO update mechanism)
def update_position(wolf, alpha, beta, delta, a):
 # Generate a new wolf position by exploring around alpha, beta, delta wolves
new_wolf = wolf.copy()

 for i in range(len(wolf)):
 r1 = np.random.rand()
 r2 = np.random.rand()

 # Move the wolf toward the alpha, beta, or delta
 if r1 < 0.33:
 new_wolf[i] = alpha[i] # Move toward alpha wolf
 elif r1 < 0.66:
 new_wolf[i] = beta[i] # Move toward beta wolf
 else:
 new_wolf[i] = delta[i] # Move toward delta wolf

 # Ensure new_wolf is a valid permutation (no duplicates)
 new_wolf = np.unique(new_wolf)
 if len(new_wolf) < len(wolf):
 # if duplicates occur, randomly swap elements to maintain uniqueness
 missing_values = list(set(range(len(wolf))) - set(new_wolf))
 np.random.shuffle(missing_values)
 new_wolf = np.append(new_wolf, missing_values)

 # Now we randomly swap some elements to increase diversity and exploration
swap_indices = np.random.choice(len(wolf), 2, replace=False)
 new_wolf[swap_indices[0]], new_wolf[swap_indices[1]] = new_wolf[swap_indices[1]],
new_wolf[swap_indices[0]]

 return new_wolf

# Main GWO loop for parking slot allocation
def gwo_parking_allocation():
 # Initialize wolves (population)
```



```
 population = initialize_population(num_wolves, num_slots)
```

```
# Variables to keep track of the best solutions
alpha = None
beta = None
delta = None

alpha_score = float('inf')
beta_score = float('inf')
delta_score = float('inf')

# GWO main loop (iterations)
for iteration in range(max_iter):
# Evaluate the fitness of each wolf in the population
fitness_scores = []
for wolf in population:
score = fitness(wolf)
fitness_scores.append(score)

# Update alpha, beta, delta based on fitness
if score < alpha_score:
delta_score, delta = beta_score, beta
beta_score, beta = alpha_score, alpha
alpha_score, alpha = score, wolf
elif score < beta_score:
delta_score, delta = beta_score, beta
beta_score, beta = score, wolf
elif score < delta_score:
delta_score, delta = score, wolf

# Update positions of wolves based on best wolves (alpha, beta, delta)
new_population = []
for wolf in population:
# Adjust the exploration factor "a" to control the step size over iterations  a = 2
- iteration * (2 / max_iter)
new_wolf = update_position(wolf, alpha, beta, delta, a)
new_population.append(new_wolf)
population = new_population

# Print progress
print(f"Iteration {iteration+1}: Best total walking distance = {alpha_score}")
# Final best result
print("\nBest Assignment of Cars to Slots (car i -> slot number):")
print(alpha)
print("Slot distances:", slot_distances[alpha])
print("Total walking distance:", alpha_score)
```

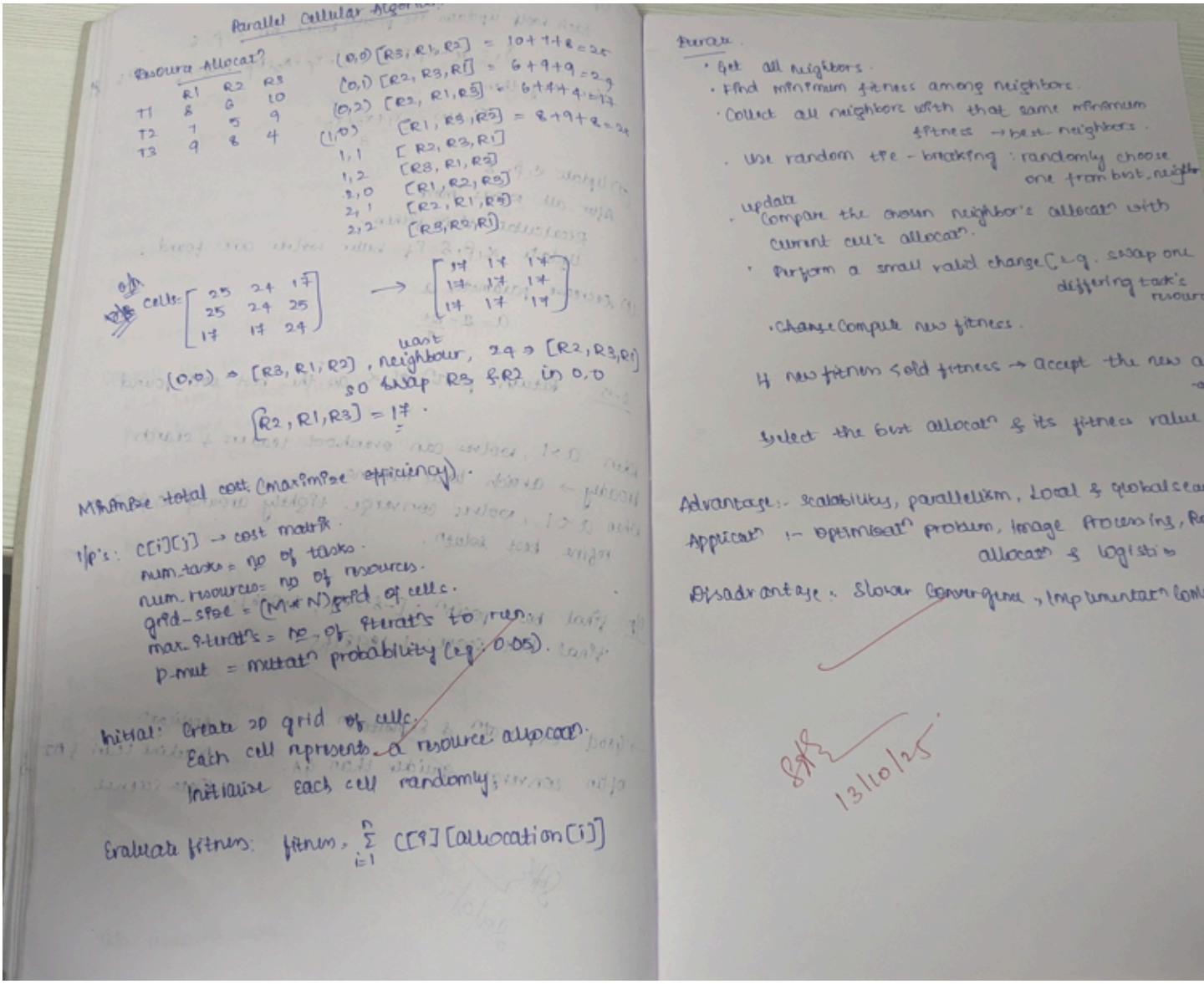


```
# Run the optimizer
gwo_parking_allocation()
```

Program 7

using **Parallel Cellular Algorithm** for resource allocation

Algorithm:



Code:

```
import random

# ---------------------------
# Cost matrix (task-resource)
# ---------------------------
```

```
C = [
    [8, 6, 10],  # Task 1 costs
    [7, 5, 9],   # Task 2 costs
    [9, 8, 4]    # Task 3 costs
]

num_tasks = len(C)
num_resources = len(C[0])

# ----------------------------
# PCA Parameters
# ----------------------------
M, N = 3, 3          # Grid size
max_iter = 5         # Number of iterations
p_mut = 0.1          # Mutation probability

# ----------------------------
# Helper Functions
# ----------------------------
def compute_cost(allocation):
    """Compute total cost for a given allocation."""
    return sum(C[i][allocation[i]] for i in range(num_tasks))

def random_allocation():
    """Generate a random valid allocation (permutation)."""
    alloc = list(range(num_resources))
    random.shuffle(alloc)
    return alloc

def random_swap(alloc):
    """Randomly swap two tasks' resource assignments."""
    a = alloc[:]
    i, j = random.sample(range(num_tasks), 2)
    a[i], a[j] = a[j], a[i]
    return a

def swap_first_difference(a, b):
    """Move a slightly toward b by swapping the first differing task."""
    a = a[:]
    for i in range(num_tasks):
        if a[i] != b[i]:
            # Find index where b[i] currently exists in a
            j = a.index(b[i])
            a[i], a[j] = a[j], a[i]
            break
    return a

def get_neighbors(grid, x, y):
```

```
    """Return Moore neighborhood of (x,y)."""
    neighbors = []
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == 0 and dy == 0:
                continue
            nx, ny = x + dx, y + dy
            if 0 <= nx < M and 0 <= ny < N:
                neighbors.append(grid[nx][ny])
    return neighbors

# ----------------------------
# Initialize grid
# ----------------------------
grid = []
for i in range(M):
    row = []
    for j in range(N):
        alloc = random_allocation()
        cost = compute_cost(alloc)
        row.append({"alloc": alloc, "cost": cost})
    grid.append(row)

# Track global best
best_cell = min((grid[i][j] for i in range(M) for j in range(N)), key=lambda c: c["cost"])
best_alloc = best_cell["alloc"][:]
best_cost = best_cell["cost"]

# ----------------------------
# Main PCA Loop
# ----------------------------
for iteration in range(1, max_iter + 1):
    new_grid = [[cell.copy() for cell in row] for row in grid]

    for i in range(M):
        for j in range(N):
            cell = grid[i][j]
            neighbors = get_neighbors(grid, i, j)

            # Find best neighbor (tie-break randomly)
            min_cost = min(n["cost"] for n in neighbors)
            best_neighbors = [n for n in neighbors if n["cost"] == min_cost]
            chosen = random.choice(best_neighbors)

            # Move slightly toward best neighbor
            candidate = swap_first_difference(cell["alloc"], chosen["alloc"])
            candidate_cost = compute_cost(candidate)
```

```
            # Mutation
            if random.random() < p_mut:
                candidate = random_swap(candidate)
                candidate_cost = compute_cost(candidate)

            # Accept if better
            if candidate_cost < cell["cost"]:
                new_grid[i][j] = {"alloc": candidate, "cost": candidate_cost}

    # Update grid
    grid = new_grid

    # Update global best
    current_best = min((grid[i][j] for i in range(M) for j in range(N)), key=lambda c: c["cost"])
    if current_best["cost"] < best_cost:
        best_cost = current_best["cost"]
        best_alloc = current_best["alloc"][:]

    # Print iteration summary
    print(f"\nIteration {iteration}:")
    for r in range(M):
        for c in range(N):
            print(grid[r][c]["alloc"], "->", grid[r][c]["cost"], end="   ")
        print()
    print("Best so far:", best_alloc, "Cost:", best_cost)

# ---------------------------
# Final Result
# ---------------------------
print("\nFinal Best Allocation:", best_alloc)
print("Minimum Total Cost:", best_cost)
```