VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

SHREYAS SINHA (1BM23CS321)

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 **Shreyas Sinha (1BM23CS321),** 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.

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

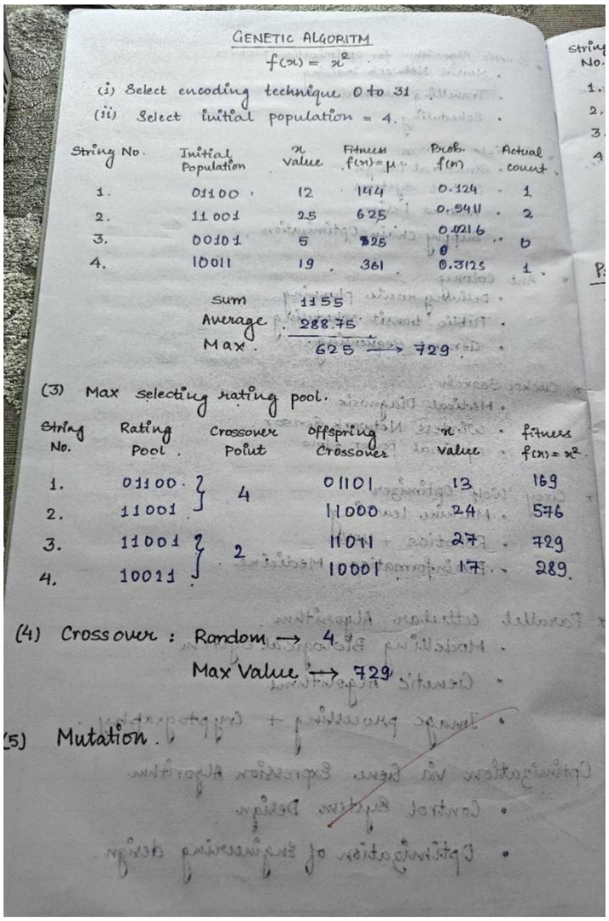
Index

GitHub Link: https://github.com/Shreyas-2607/BIS_LAB

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



GENETIC ALGORITM

$f(x) = x^2$

(i) Select encoding technique 0 to 31

(ii) Select initial population = 4.

String No.	Initial Population	x value	Fitness $f(x) = \mu$	Prob. $f(x)$	Actual count
1.	01100	12	144	0.124	1
2.	11001	25	625	0.5411	2
3.	00101	5	25	0.0216	0
4.	10011	19	361	0.3125	1

Sum 1155

Average 288.75

Max 625 → 729

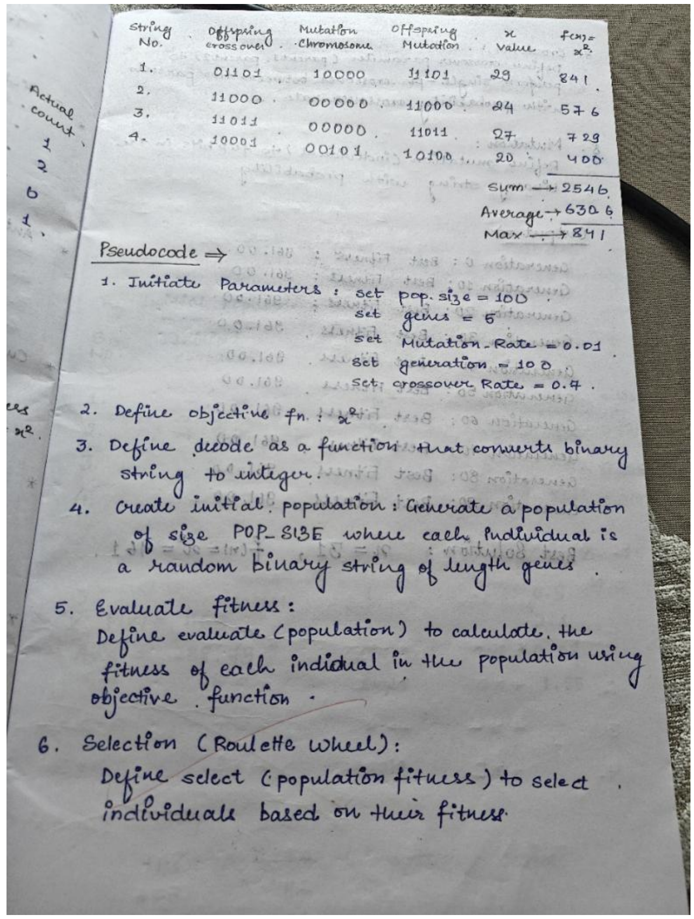
(3) Max selecting rating pool.

String No.	Rating Pool	Crossover Point	Offspring Crossover	x value	fitness $f(x) = x^2$
1.	01100	4	01101	13	169
2.	11001		11000	24	576
3.	11001	2	11011	27	729
4.	10011		10001	17	289

(4) Crossover : Random → 4

Max Value → 729

(5) Mutation.



String No.	Offspring crossover	Mutation Chromosome	Offspring Mutation	x Value	$f(x) = x^2$
1.	01101	10000	11101	29	841
2.	11000	00000	11000	24	576
3.	11011	00000	11011	27	729
4.	10001	00101	10100	20	400

Sum → 2546

Average → 630.6

Max → 841

Pseudocode ⇒

1. Initiate Parameters : set pop. size = 100
   set genes = 5
   set Mutation_Rate = 0.01
   set generation = 100
   set crossover Rate = 0.7
2. Define objective fn. : x^2
3. Define decode as a function that converts binary string to integer.
4. Create initial population : Generate a population of size POP_SIZE where each individual is a random binary string of length genes.
5. Evaluate fitness :
   Define evaluate (population) to calculate the fitness of each indidual in the population using objective function.
6. Selection (Roulette wheel):
   Define select (population fitness) to select individuals based on their fitness.

Code:

```
import random

def fitness_function(x):
    return x ** 2

def decode(chromosome):
    return int(chromosome, 2)

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

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    if total_fitness == 0:
        return random.choice(population)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, CHROMOSOME_LENGTH - 1)
        return (parent1[:point] + parent2[point:], parent2[:point] + parent1[point:])
    return parent1, parent2

def mutate(chromosome):
    new_chromosome = ''
    for bit in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome += '0' if bit == '1' else '1'
        else:
            new_chromosome += bit
    return new_chromosome

def get_initial_population(size, length):
    population = []
    print(f"Enter {size} chromosomes (each of {length} bits, e.g., '10101'):")
    while len(population) < size:
        chrom = input(f"Chromosome {len(population)+1}: ").strip()
        if len(chrom) == length and all(bit in '01' for bit in chrom):
            population.append(chrom)
        else:
            print(f"Invalid input. Please enter a {length}-bit binary string.")
```

```
    return population

def genetic_algorithm():
    population = get_initial_population(POPULATION_SIZE, CHROMOSOME_LENGTH)
    best_solution = None
    best_fitness = float('-inf')

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, individual in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = individual

        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best x = 
{decode(best_solution)}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Chromosome: {best_solution}")
    print(f"x = {decode(best_solution)}")
    print(f"f(x) = {fitness_function(decode(best_solution))}")

POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5
MUTATION_RATE = 0.01
CROSSOVER_RATE = 0.8
GENERATIONS = 20

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

Output:

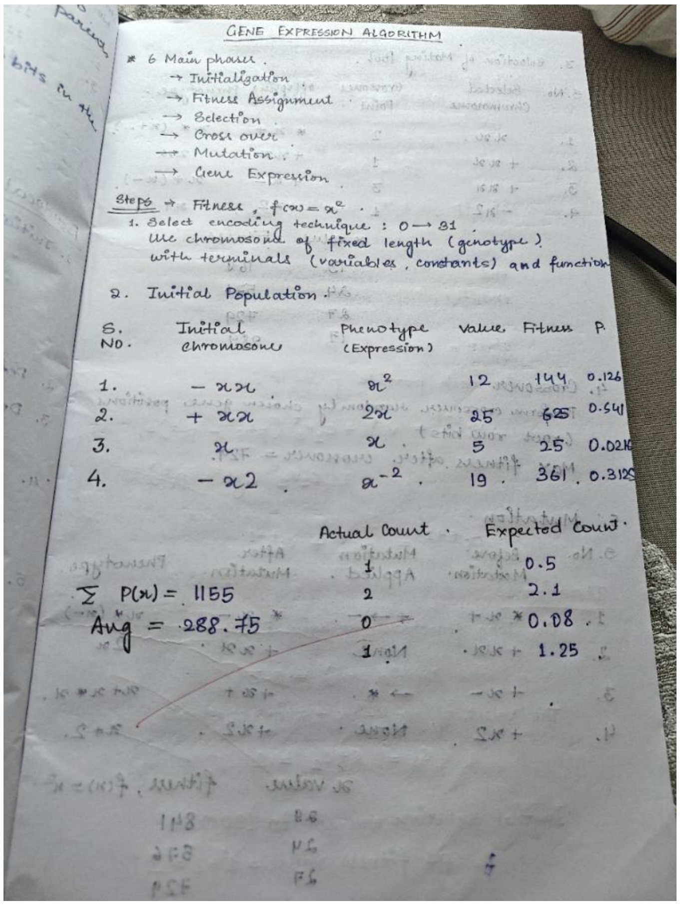
```
Enter 4 chromosomes (each of 5 bits, e.g., '10101'):
Chromosome 1: 01100
Chromosome 2: 11001
Chromosome 3: 00101
Chromosome 4: 10011
Generation 1: Best Fitness = 625, Best x = 25
Generation 2: Best Fitness = 625, Best x = 25
Generation 3: Best Fitness = 625, Best x = 25
Generation 4: Best Fitness = 625, Best x = 25
Generation 5: Best Fitness = 625, Best x = 25
Generation 6: Best Fitness = 625, Best x = 25
Generation 7: Best Fitness = 625, Best x = 25
Generation 8: Best Fitness = 625, Best x = 25
Generation 9: Best Fitness = 625, Best x = 25
Generation 10: Best Fitness = 625, Best x = 25
Generation 11: Best Fitness = 625, Best x = 25
Generation 12: Best Fitness = 625, Best x = 25
Generation 13: Best Fitness = 625, Best x = 25
Generation 14: Best Fitness = 625, Best x = 25
Generation 15: Best Fitness = 625, Best x = 25
Generation 16: Best Fitness = 625, Best x = 25
Generation 17: Best Fitness = 625, Best x = 25
Generation 18: Best Fitness = 625, Best x = 25
Generation 19: Best Fitness = 625, Best x = 25
Generation 20: Best Fitness = 625, Best x = 25

Best solution found:
Chromosome: 11001
x = 25
f(x) = 625
```

Program 2

Optimization via Gene Expression Algorithms:

Algorithm:



GENE EXPRESSION ALGORITHM

* 6 Main phases.
  - → Initialization
  - → Fitness Assignment
  - → Selection
  - → Cross over
  - → Mutation
  - → Gene Expression

Steps → Fitness, $f(x) = x^2$.

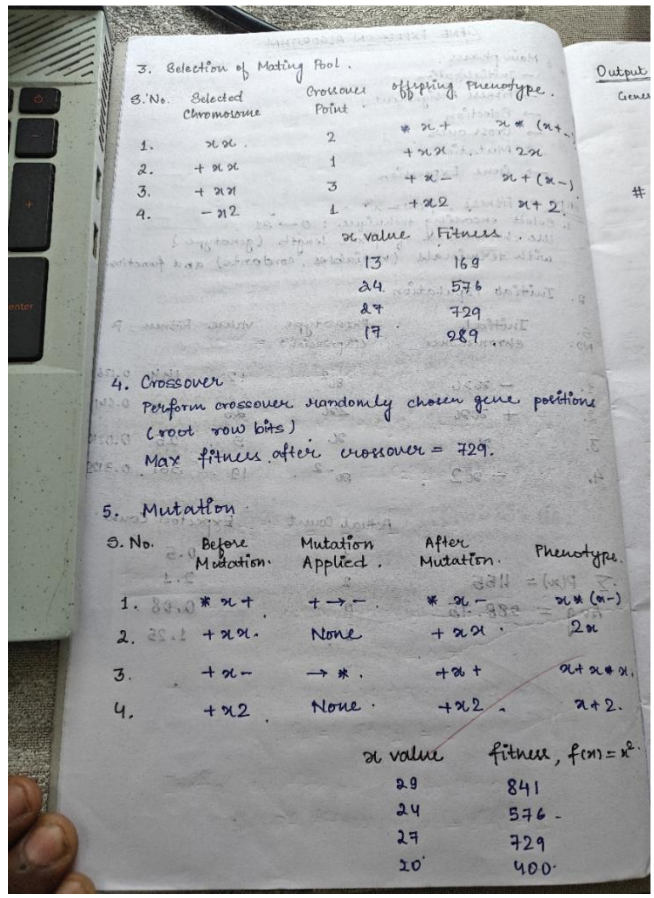
1. Select encoding technique : 0 → 31
   We chromosome of fixed length (genotype) with terminals (variables, constants) and function
2. Initial Population

S. No.	Initial chromosome	Phenotype (Expression)	Value	Fitness	P
1.	$-xx$	x^2	12	144	0.126
2.	$+xx$	$2x$	25	625	0.541
3.	x	x	5	25	0.0216
4.	$-x2$	x^{-2}	19	361	0.3125

Actual Count	Expected Count
1	0.5
2	2.1
0	0.08
1	1.25

$\Sigma P(x) = 1155$

Avg = 288.75



3. Selection of Mating Pool.

S. No.	Selected Chromosome	Crossover Point	offspring	Phenotype
1.	xx	2	$*x+$	$x*(x+)$
2.	$+xx$	1	$+xx$	$2x$
3.	$+xx$	3	$+x-$	$x+(x-)$
4.	$-x2$	1	$+x2$	$x+2$

x value	Fitness
13	169
24	576
27	729
17	289

4. Crossover

Perform crossover randomly chosen gene positions (root row bits).

Max fitness after crossover = 729.

5. Mutation

S. No.	Before Mutation	Mutation Applied	After Mutation	Phenotype
1.	$*x+$	$+ \to -$	$*x-$	$x*(x-)$
2.	$+xx$	None	$+xx$	$2x$
3.	$+x-$	$\to *$	$+x+$	$x+x*x$
4.	$+x2$	None	$+x2$	$x+2$

x value	fitness, $f(x) = x^2$
29	841
24	576
27	729
20	400

Code:

```
import random
import math

def fitness_function(x):
    return x * math.sin(10 * math.pi * x) + 2

POPULATION_SIZE = 6
GENE_LENGTH = 10
MUTATION_RATE = 0.05
CROSSOVER_RATE = 0.8
GENERATIONS = 20
DOMAIN = (-1, 2)
```

```
def random_gene():
    return random.uniform(DOMAIN[0], DOMAIN[1])

def create_chromosome():
    return [random_gene() for _ in range(GENE_LENGTH)]

def initialize_population(size):
    return [create_chromosome() for _ in range(size)]

def evaluate_population(population):
    return [fitness_function(express_gene(chrom)) for chrom in population]

def express_gene(chromosome):
    return sum(chromosome) / len(chromosome)

def select(population, fitnesses):
    total_fitness = sum(fitnesses)
    pick = random.uniform(0, total_fitness)
    current = 0
    for individual, fitness in zip(population, fitnesses):
        current += fitness
        if current > pick:
            return individual
    return random.choice(population)

def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GENE_LENGTH - 1)
        child1 = parent1[:point] + parent2[point:]
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1[:], parent2[:]

def mutate(chromosome):
    new_chromosome = []
    for gene in chromosome:
        if random.random() < MUTATION_RATE:
            new_chromosome.append(random_gene())
        else:
            new_chromosome.append(gene)
    return new_chromosome

def gene_expression_algorithm():
    population = initialize_population(POPULATION_SIZE)
    best_solution = None
    best_fitness = float("-inf")
```

```
    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        for i, chrom in enumerate(population):
            if fitnesses[i] > best_fitness:
                best_fitness = fitnesses[i]
                best_solution = chrom[:]

        print(f"Generation {generation+1}: Best Fitness = {best_fitness:.4f}, Best x = 
{express_gene(best_solution):.4f}")

        new_population = []
        while len(new_population) < POPULATION_SIZE:
            parent1 = select(population, fitnesses)
            parent2 = select(population, fitnesses)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)
            new_population.extend([offspring1, offspring2])

        population = new_population[:POPULATION_SIZE]

    print("\nBest solution found:")
    print(f"Genes: {best_solution}")
    x_value = express_gene(best_solution)
    print(f"x = {x_value:.4f}")
    print(f"f(x) = {fitness_function(x_value):.4f}")

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

Output:

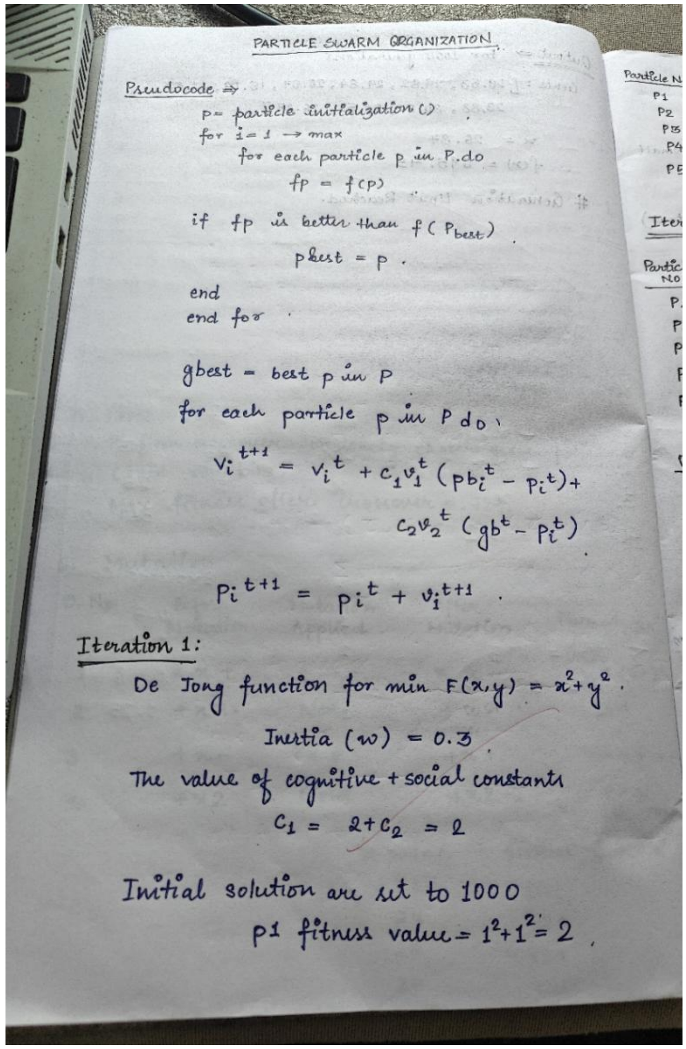
```
Generation 1: Best Fitness = 2.3125, Best x = 0.4262
Generation 2: Best Fitness = 2.3125, Best x = 0.4262
Generation 3: Best Fitness = 2.3125, Best x = 0.4262
Generation 4: Best Fitness = 2.3125, Best x = 0.4262
Generation 5: Best Fitness = 2.3125, Best x = 0.4262
Generation 6: Best Fitness = 2.3125, Best x = 0.4262
Generation 7: Best Fitness = 2.3125, Best x = 0.4262
Generation 8: Best Fitness = 2.4233, Best x = 0.6237
Generation 9: Best Fitness = 2.4233, Best x = 0.6237
Generation 10: Best Fitness = 2.4233, Best x = 0.6237
Generation 11: Best Fitness = 2.4233, Best x = 0.6237
Generation 12: Best Fitness = 2.4233, Best x = 0.6237
Generation 13: Best Fitness = 2.4233, Best x = 0.6237
Generation 14: Best Fitness = 2.4233, Best x = 0.6237
Generation 15: Best Fitness = 2.4233, Best x = 0.6237
Generation 16: Best Fitness = 2.4233, Best x = 0.6237
Generation 17: Best Fitness = 2.4395, Best x = 0.4594
Generation 18: Best Fitness = 2.4395, Best x = 0.4594
Generation 19: Best Fitness = 2.4395, Best x = 0.4594
Generation 20: Best Fitness = 2.4395, Best x = 0.4594

Best solution found:
Genes: [0.6948405045559576, -0.647173288232043, -0.3013499383055478, 1.631627548910124, 0.9271637073163099, 0.0324867196364278, -0.3565755055362756, 1.5226396608397925, 1.0654293190513275
x = 0.4594
f(x) = 2.4395
```

Program 3

Particle Swarm Optimization

Algorithm:



PARTICLE SWARM ORGANIZATION

Pseudocode ⇒

```
p = particle initialization ()
for i = 1 → max
    for each particle p in P.do
        fp = f(p)
if fp is better than f(P_best)
        pbest = p.
end
end for

gbest = best p in P
for each particle p in P do
```

$$v_i^{t+1} = v_i^t + c_1 v_1^t (pb_i^t - p_i^t) + c_2 v_2^t (gb^t - p_i^t)$$

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

Iteration 1:

De Jong function for min $F(x,y) = x^2 + y^2$.

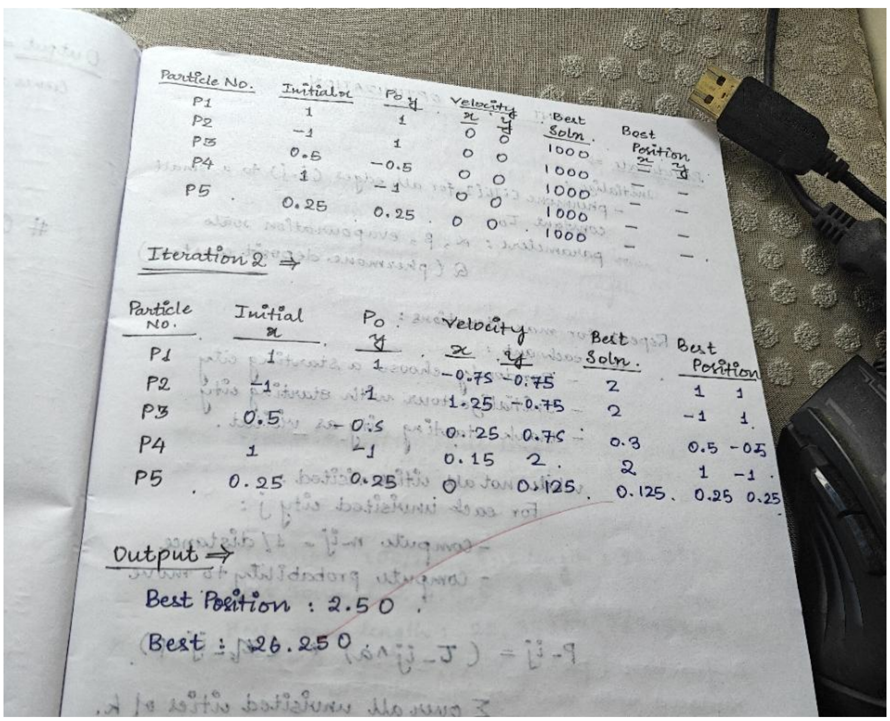
Inertia (w) = 0.3

The value of cognitive + social constants

$c_1 = 2 + c_2 = 2$

Initial solution are set to 1000

p1 fitness value = $1^2 + 1^2 = 2$.



Particle No.	Initial x	Po y	Velocity x	Velocity y	Best Soln.	Best Position x	Best Position y
P1	1	1	0	0	1000	—	—
P2	−1	1	0	0	1000	—	—
P3	0.5	−0.5	0	0	1000	—	—
P4	1	−1	0	0	1000	—	—
P5	0.25	0.25	0	0	1000	—	—

Iteration 2 ⇒

Particle No.	Initial x	Po y	Velocity x	Velocity y	Best Soln.	Best Position x	Best Position y
P1	1	1	−0.75	−0.75	2	1	1
P2	−1	1	1.25	−0.75	2	−1	1
P3	0.5	−0.5	0.25	0.75	0.3	0.5	−0.5
P4	1	−1	0.15	2	2	1	−1
P5	0.25	0.25	0	0.125	0.125	0.25	0.25

Output ⇒

Best Position : 2.50.

Best : 26.250

Code:

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


def objective_function(x):
    return x**2 - 4*x + 4

class Particle:
    def __init__(self, lower_bound, upper_bound):
        self.position = np.random.uniform(lower_bound, upper_bound)
        self.velocity = np.random.uniform(-1, 1)
        self.best_position = self.position
        self.best_value = objective_function(self.position)

    def update(self, global_best_position, w, c1, c2):
        r1 = np.random.rand()
        r2 = np.random.rand()


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


        current_value = objective_function(self.position)


        if current_value < self.best_value:
            self.best_value = current_value
            self.best_position = self.position

# PSO parameters
num_particles = 30
num_iterations = 100
w = 0.7
c1 = 1.5
c2 = 1.5
lower_bound = -10
upper_bound = 10


particles = [Particle(lower_bound, upper_bound) for _ in range(num_particles)]


global_best_position = particles[0].best_position
global_best_value = particles[0].best_value
```

```
# PSO loop
for iteration in range(num_iterations):
    for particle in particles:
        particle.update(global_best_position, w, c1, c2)


        if particle.best_value < global_best_value:
            global_best_value = particle.best_value
            global_best_position = particle.best_position


    if iteration % 10 == 0:
        print(f"Iteration {iteration}: Global Best Position = {global_best_position}, Value =
{global_best_value}")


print(f"\nOptimal Position: {global_best_position}")
print(f"Optimal Value: {global_best_value}")


x_values = np.linspace(lower_bound, upper_bound, 100)
y_values = objective_function(x_values)

plt.plot(x_values, y_values, label="Objective Function f(x)")
plt.scatter(global_best_position, global_best_value, color='red', label='Optimal Solution (PSO)', zorder=5)
plt.title("Particle Swarm Optimization for Minimizing f(x) = x^2 - 4x + 4")
plt.xlabel("x")
plt.ylabel("f(x)")
plt.legend()
plt.show()
```

Output:

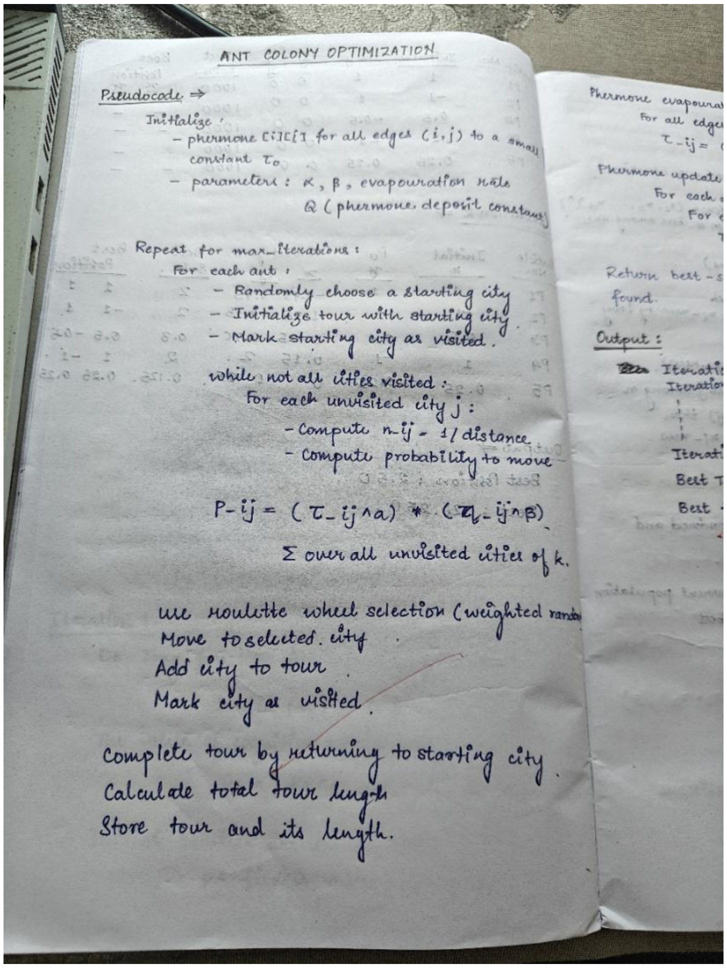
```
••• Iteration 2:
        x     y    vx     vy  pbest_x  pbest_y  pbest_val
    0  1.00  1.00 -0.75 -0.750    1.00     1.00     27.000
    1 -1.00  1.00  1.25 -0.750   -1.00     1.00     27.000
    2  0.50 -0.50  0.25  0.750    0.50    -0.50     25.500
    3  1.00 -1.00  0.15  2.000    1.00    -1.00     27.000
    4  0.85  0.25  0.00  0.125    0.85     0.25     25.785

    Output:
    Best position: (0.5, -0.5)
    Best value: 25.500
```

Program 4

Ant Colony Optimization for Vehicle Routing Problem:

Algorithm:



ANT COLONY OPTIMIZATION

Pseudocode ⇒

Initialize :

- phermone [i][j] for all edges (i, j) to a small constant τ_0
- parameters : α, β, evapouration rate
  Q (phermone deposit constant)

Repeat for max_iterations :

For each ant :

- Randomly choose a starting city
- Initialize tour with starting city .
- Mark starting city as visited .

while not all cities visited :

For each unvisited city j :

- Compute n_ij = 1/ distance
- Compute probability to move

P_ij = (τ_ij ^a) * (η_ij ^ β)

Σ over all unvisited cities of k.

use roulette wheel selection (weighted random)

Move to selected city .

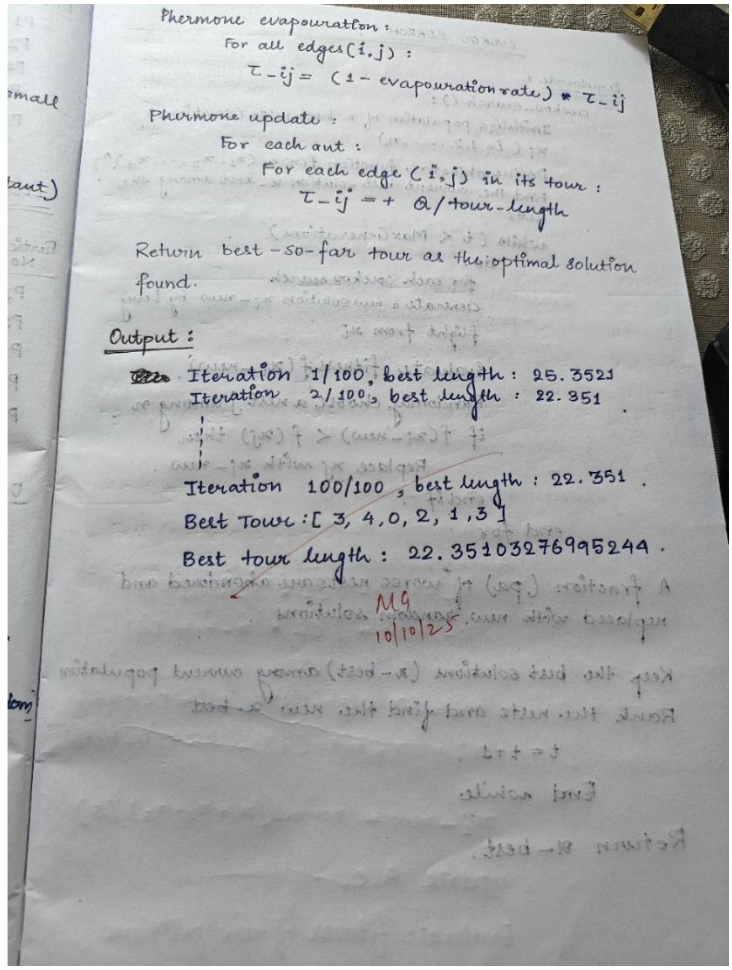
Add city to tour .

Mark city as visited .

Complete tour by returning to starting city .

Calculate total tour length

Store tour and its length.



Phermone evapouration :

For all edges(i, j) :

τ_ij = (1 - evapouration rate) * τ_ij

Phermone update :

For each ant :

For each edge (i, j) in its tour :

τ_ij =+ Q/tour_length

Return best-so-far tour as the optimal solution found.

Output :

Iteration 1/100, best length : 25.3521

Iteration 2/100, best length : 22.351 .

Iteration 100/100 , best length : 22.351 .

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

Best tour length : 22.35103276995244 .

MG
10/10/25

Code:

```
import random
import numpy as np

class VRP:
    def __init__(self, depot, customers, capacities):
        self.depot = depot
        self.customers = customers
        self.capacities = capacities
        self.num_customers = len(customers)
        self.num_vehicles = len(capacities)
        self.distance_matrix = self.create_distance_matrix()

    def create_distance_matrix(self):
        dist_matrix = np.zeros((self.num_customers + 1, self.num_customers + 1))  # +1 for depot
        for i in range(self.num_customers + 1):
            for j in range(i + 1, self.num_customers + 1):
                if i == 0:
                    dist = np.linalg.norm(np.array(self.depot) - np.array(self.customers[j-1]))
                elif j == 0:
                    dist = np.linalg.norm(np.array(self.depot) - np.array(self.customers[i-1]))
                else:
                    dist = np.linalg.norm(np.array(self.customers[i-1]) - np.array(self.customers[j-1]))
                dist_matrix[i][j] = dist_matrix[j][i] = dist
        return dist_matrix

class AntColony:
    def __init__(self, vrp, num_ants, alpha=1, beta=5, rho=0.5, q0=0.9, iterations=100):
        self.vrp = vrp
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q0 = q0
        self.iterations = iterations
        self.pheromone = np.ones((vrp.num_customers + 1, vrp.num_customers + 1))
        self.best_solution = None
        self.best_solution_length = float('inf')

    def construct_solution(self, ant_idx):
        unvisited = set(range(1, self.vrp.num_customers + 1))
        routes = {vehicle: [] for vehicle in range(self.vrp.num_vehicles)}
        demands = {vehicle: 0 for vehicle in range(self.vrp.num_vehicles)}
        current_city = 0

        while unvisited:
            vehicle = random.choice(range(self.vrp.num_vehicles))
            if demands[vehicle] < self.vrp.capacities[vehicle]:
                next_city = self.select_next_city(current_city, unvisited)
                routes[vehicle].append(next_city)
                unvisited.remove(next_city)
```

```
            demands[vehicle] += 1
            current_city = next_city
        else:
            continue

    for vehicle in range(self.vrp.num_vehicles):
        routes[vehicle].append(0)

    return routes

def calculate_probabilities(self, current_city, unvisited):
    probabilities = []
    for city in unvisited:
        pheromone = self.pheromone[current_city][city] ** self.alpha
        distance = self.vrp.distance_matrix[current_city][city]
        heuristic = (1 / distance) ** self.beta
        probabilities.append(pheromone * heuristic)

    total_prob = sum(probabilities)
    if total_prob == 0:
        return [1 / len(unvisited)] * len(unvisited)

    probabilities = [p / total_prob for p in probabilities]
    return probabilities

def select_next_city(self, current_city, unvisited):
    probabilities = self.calculate_probabilities(current_city, unvisited)
    return random.choices(list(unvisited), probabilities)[0]

def update_pheromones(self, solutions, lengths):
    self.pheromone *= (1 - self.rho)

    for idx, solution in enumerate(solutions):
        length = lengths[idx]
        for route in solution.values():
            for i in range(len(route) - 1):
                self.pheromone[route[i]][route[i + 1]] += 1 / length

def run(self):
    for iteration in range(self.iterations):
        all_solutions = []
        all_lengths = []

        for ant_idx in range(self.num_ants):
            solution = self.construct_solution(ant_idx)
            length = self.calculate_solution_length(solution)
            all_solutions.append(solution)
            all_lengths.append(length)

            if length < self.best_solution_length:
                self.best_solution = solution
                self.best_solution_length = length
```

```
            self.update_pheromones(all_solutions, all_lengths)

            print(f"Iteration {iteration + 1}/{self.iterations}: Best Length = {self.best_solution_length}")

        return self.best_solution, self.best_solution_length

    def calculate_solution_length(self, solution):
        length = 0
        for vehicle in solution.values():
            for i in range(len(vehicle) - 1):
                length += self.vrp.distance_matrix[vehicle[i]][vehicle[i + 1]]
        return length

if __name__ == "__main__":
    depot = (0, 0)
    customers = [(2, 4), (3, 2), (6, 5), (8, 3), (7, 8), (5, 7)]
    capacities = [3, 3]

    vrp = VRP(depot, customers, capacities)
    aco = AntColony(vrp, num_ants=10, alpha=1, beta=2, rho=0.5, q0=0.9, iterations=50)

    best_solution, best_solution_length = aco.run()

    print("Best Solution (Routes):")
    for vehicle, route in best_solution.items():
        print(f"Vehicle {vehicle + 1}: {route}")

    print(f"\nBest Solution Length (Total Distance): {best_solution_length}")
```

Output:

```
Iteration 1/50: Best Length = 30.704096057970965
Iteration 2/50: Best Length = 25.85324233796014
Iteration 3/50: Best Length = 25.85324233796014
Iteration 4/50: Best Length = 25.85324233796014
Iteration 5/50: Best Length = 25.85324233796014
Iteration 6/50: Best Length = 25.85324233796014
Iteration 7/50: Best Length = 25.80789435080295
Iteration 8/50: Best Length = 25.80789435080295
Iteration 9/50: Best Length = 25.80789435080295
Iteration 10/50: Best Length = 25.80789435080295
Iteration 11/50: Best Length = 25.738177938973646
Iteration 12/50: Best Length = 25.738177938973646
Iteration 13/50: Best Length = 25.738177938973646
Iteration 14/50: Best Length = 22.65045276546171
Iteration 15/50: Best Length = 22.65045276546171
Iteration 16/50: Best Length = 22.65045276546171
Iteration 17/50: Best Length = 22.65045276546171
Iteration 18/50: Best Length = 22.65045276546171
Iteration 19/50: Best Length = 22.65045276546171
Iteration 20/50: Best Length = 22.65045276546171
Iteration 21/50: Best Length = 22.65045276546171
Iteration 22/50: Best Length = 22.65045276546171
Iteration 23/50: Best Length = 22.483842533421615
Iteration 24/50: Best Length = 22.483842533421615
Iteration 25/50: Best Length = 22.483842533421615
Iteration 26/50: Best Length = 22.483842533421615
Iteration 27/50: Best Length = 22.483842533421615
Iteration 28/50: Best Length = 22.483842533421615
Iteration 29/50: Best Length = 22.483842533421615
Iteration 30/50: Best Length = 22.483842533421615
Iteration 31/50: Best Length = 22.483842533421615
Iteration 32/50: Best Length = 22.483842533421615
Iteration 33/50: Best Length = 22.483842533421615
Iteration 34/50: Best Length = 22.483842533421615
Iteration 35/50: Best Length = 22.483842533421615
```

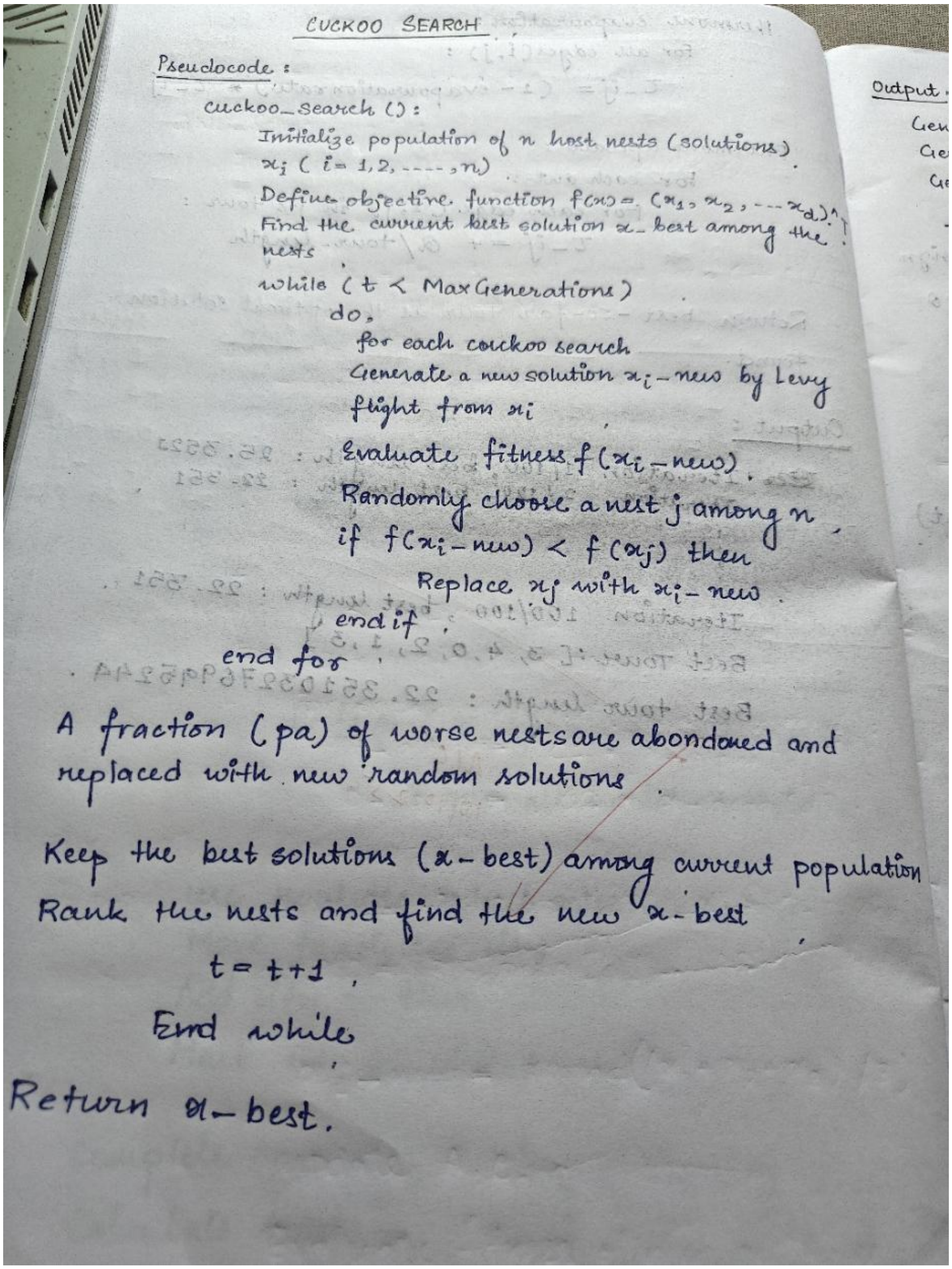
```
Iteration 36/50: Best Length = 22.483842533421615
Iteration 37/50: Best Length = 22.483842533421615
Iteration 38/50: Best Length = 22.483842533421615
Iteration 39/50: Best Length = 22.483842533421615
Iteration 40/50: Best Length = 22.483842533421615
Iteration 41/50: Best Length = 22.483842533421615
Iteration 42/50: Best Length = 22.483842533421615
Iteration 43/50: Best Length = 22.483842533421615
Iteration 44/50: Best Length = 22.483842533421615
Iteration 45/50: Best Length = 22.483842533421615
Iteration 46/50: Best Length = 22.483842533421615
Iteration 47/50: Best Length = 22.483842533421615
Iteration 48/50: Best Length = 22.483842533421615
Iteration 49/50: Best Length = 22.483842533421615
Iteration 50/50: Best Length = 22.483842533421615
Best Solution (Routes):
Vehicle 1: [5, 6, 1, 0]
Vehicle 2: [3, 4, 2, 0]

Best Solution Length (Total Distance): 22.483842533421615
```

Program 5

Cuckoo Search Algorithm for Task Scheduling Optimization:

Algorithm:



CUCKOO SEARCH

Pseudocode :

```
cuckoo_search ():
    Initialize population of n host nests (solutions)
    xi ( i= 1,2, ---- ,n)
    Define objective function f(x) = (x1, x2, --- xd)^T
    Find the current best solution x_best among the nests
    while ( t < MaxGenerations )
        do,
            for each cuckoo search
            Generate a new solution xi-new by Levy
            flight from xi
            Evaluate fitness f(xi-new)
            Randomly choose a nest j among n
            if f(xi-new) < f(xj) then
                Replace xj with xi-new
            end if
        end for
A fraction (pa) of worse nests are abondoned and
replaced with new random solutions

Keep the best solutions (x-best) among current population
Rank the nests and find the new x-best
        t = t+1
    End while
Return x-best.
```

Code:

```
import random

tasks = [2, 3, 4, 5, 6]
num_tasks = len(tasks)
num_nests = 5
Pa = 0.25
MaxGen = 50

def fitness(schedule):
    """Lower total duration = better fitness."""
    total_time = 0
    for t in schedule:
        total_time += t
    return -total_time

def random_schedule():
    """Create a random schedule (random order of tasks)."""
    s = tasks[:]
    random.shuffle(s)
    return s

def levy_flight(schedule):
    """Generate new schedule by small random changes."""
    new_s = schedule[:]
    i, j = random.sample(range(num_tasks), 2)
    new_s[i], new_s[j] = new_s[j], new_s[i]
    return new_s

nests = [random_schedule() for _ in range(num_nests)]
fitness_values = [fitness(s) for s in nests]

for gen in range(MaxGen):
    for i in range(num_nests):
        new_solution = levy_flight(nests[i])
        new_fitness = fitness(new_solution)

        j = random.randint(0, num_nests - 1)
        if new_fitness > fitness_values[j]:
            nests[j] = new_solution
            fitness_values[j] = new_fitness

    sorted_nests = sorted(zip(fitness_values, nests), reverse=True)
    num_abandon = int(Pa * num_nests)
    for k in range(num_abandon):
        sorted_nests[-(k+1)] = (fitness(random_schedule()), random_schedule())

    fitness_values, nests = zip(*sorted_nests)
```

```
        fitness_values, nests = list(fitness_values), list(nests)

    best_index = fitness_values.index(max(fitness_values))
    print("Best Schedule:", nests[best_index])
    print("Total Time:", -fitness_values[best_index])
```

Output:

```
Best Schedule: [6, 5, 4, 3, 2]
Total Time: 20
```

Program 6

Grey Wolf Optimizer (GWO):

Algorithm:

GREY WOLF OPTIMIZER

Application ⇒ Scheduling and Resource Allocation.

GWO_Scheduling () :

Initialize population of wolves (solutions).

x_i for $i = 1, \ldots, n$.

Each x_i represents a task-to-machine assignment.

Ex: $x_i = [0, 1, 2, 1, 0]$ means task 0 → M0, task 1 ⇒ M1 etc.

Evaluate fitness for each x_i

- for each solution, complete makespan (maximum machine time).

Identify Alpha (best), Beta (second best), Delta wolves (third best).

while (t < MaxIterations)

For each wolf x_i in population :

For each dimension d (task assignment) :

D-alpha = | C1 * x-alpha [d] |

D-beta = | C2 * x-beta [d] .

x1 = x-alpha [d] - A1 * D-alpha

x2 = x-beta [d] - A2 * D-beta

x_i-new = round(($x_1 + x_2 + x_3$) / 3)

update A, C, a.

Evaluate fitness of new positions.

t = t+1

Return Alpha.

Code:

```
import numpy as np

num_tasks = 20
num_vms = 5
num_wolves = 15
max_iter = 50

task_load = np.random.randint(1000, 10000, num_tasks)

vm_speed = np.random.randint(500, 2000, num_vms)

def fitness(position):
    """
    position[i] = VM index assigned to task i
    Fitness = total makespan (time to finish all tasks)
    """
    loads = np.zeros(num_vms)
    for i, vm in enumerate(position.astype(int)):
        loads[vm] += task_load[i] / vm_speed[vm]
    return np.max(loads)

wolves = np.random.randint(0, num_vms, (num_wolves, num_tasks))
alpha, beta, delta = None, None, None
alpha_score, beta_score, delta_score = np.inf, np.inf, np.inf

for t in range(max_iter):
    a = 2 - 2 * t / max_iter

    for i in range(num_wolves):
        score = fitness(wolves[i])

        if score < alpha_score:
            alpha_score, alpha = score, wolves[i].copy()
        elif score < beta_score:
            beta_score, beta = score, wolves[i].copy()
        elif score < delta_score:
            delta_score, delta = score, wolves[i].copy()

    for i in range(num_wolves):
        for j in range(num_tasks):
            r1, r2 = np.random.rand(), np.random.rand()
            A1, C1 = 2*a*r1 - a, 2*r2
            D_alpha = abs(C1 * alpha[j] - wolves[i][j])
            X1 = alpha[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[j] - wolves[i][j])
            X2 = beta[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[j] - wolves[i][j])
            X3 = delta[j] - A3 * D_delta

            new_pos = (X1 + X2 + X3) / 3
            wolves[i][j] = np.clip(round(new_pos), 0, num_vms - 1)

best_allocation = alpha.astype(int)
best_makespan = alpha_score

print("Best Makespan:", best_makespan)
print("Best Task Allocation (task → VM):")
for i, vm in enumerate(best_allocation):
    print(f"  Task {i+1} → VM {vm+1}")
```

Output:

```
Best Makespan: 21.60720720720721
Best Task Allocation (task → VM):
  Task 1 → VM 2
  Task 2 → VM 1
  Task 3 → VM 4
  Task 4 → VM 5
  Task 5 → VM 3
  Task 6 → VM 5
  Task 7 → VM 4
  Task 8 → VM 1
  Task 9 → VM 1
  Task 10 → VM 5
  Task 11 → VM 4
  Task 12 → VM 3
  Task 13 → VM 4
  Task 14 → VM 2
  Task 15 → VM 3
  Task 16 → VM 5
  Task 17 → VM 4
  Task 18 → VM 2
  Task 19 → VM 1
  Task 20 → VM 4
```

Program 7

Parallel Cellular Algorithm :

Algorithm:

PARALLEL CELLULAR ALGORITHM

(i) Define objective function f(x) to be minimized or maximized.

(ii) Set algorithm parameters :
- grit size.
- num iterations.
- neighbourhood_type.
- alpha (diffusion coefficient).
- mutation_rate.

(iii) Initialize each cell with a random position, x_i in the search space.

(iv) Evaluate fitness (x_i) for each cell.

(v) For each iteration t = 1 to num_iterations :

for each cell i in the grid :
- Get neighbour cells based on neighbourhood type.
- Find best neighbour solution x_best.
- Update cell's solution :

x_to_new = x_i + alpha * (x_best - x_i) + mutation_rate * random_noise

(vi) Evaluate f(x_i_new) for all cells.

Replace x_i with x_i_new if it improves ~~or~~ fitness

vii) Output best solution found

Output ⇒

Iteration

Code:

```
import numpy as np

N = 20
infection_prob = 0.25
infection_duration = 5
steps = 10

grid = np.zeros((N, N), dtype=int)
infection_time = np.zeros((N, N), dtype=int)

for _ in range(5):
    x, y = np.random.randint(0, N, 2)
    grid[x, y] = 1

print("Step | Healthy | Infected | Recovered")
print("--------------------------------------")

for step in range(steps):
    new_grid = grid.copy()

    for i in range(N):
        for j in range(N):
            if grid[i, j] == 0:
                for dx in [-1, 0, 1]:
                    for dy in [-1, 0, 1]:
                        if dx == 0 and dy == 0:
                            continue
                        ni, nj = (i + dx) % N, (j + dy) %
N
                        if grid[ni, nj] == 1 and
np.random.rand() < infection_prob:
                            new_grid[i, j] = 1
                            break
            elif grid[i, j] == 1:
                infection_time[i, j] += 1
                if infection_time[i, j] >
infection_duration:
                    new_grid[i, j] = 2

    grid = new_grid

    healthy = np.count_nonzero(grid == 0)
    infected = np.count_nonzero(grid == 1)
    recovered = np.count_nonzero(grid == 2)
```

    print(f"{step:>4} | {healthy:>7} |
{infected:>8} | {recovered:>9}")

Output:

```
•••  Step | Healthy | Infected | Recovered
     --------------------------------------
        0 |     388 |       12 |         0
        1 |     371 |       29 |         0
        2 |     338 |       62 |         0
        3 |     301 |       99 |         0
        4 |     244 |      156 |         0
        5 |     200 |      195 |         5
        6 |     162 |      226 |        12
        7 |     106 |      265 |        29
        8 |      64 |      274 |        62
        9 |      30 |      271 |        99
```