VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

BJ KEERTANA (1BM23CS059)

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 **BJ Keertana(1BM23CS059),** 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.

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

Index

Github Link:

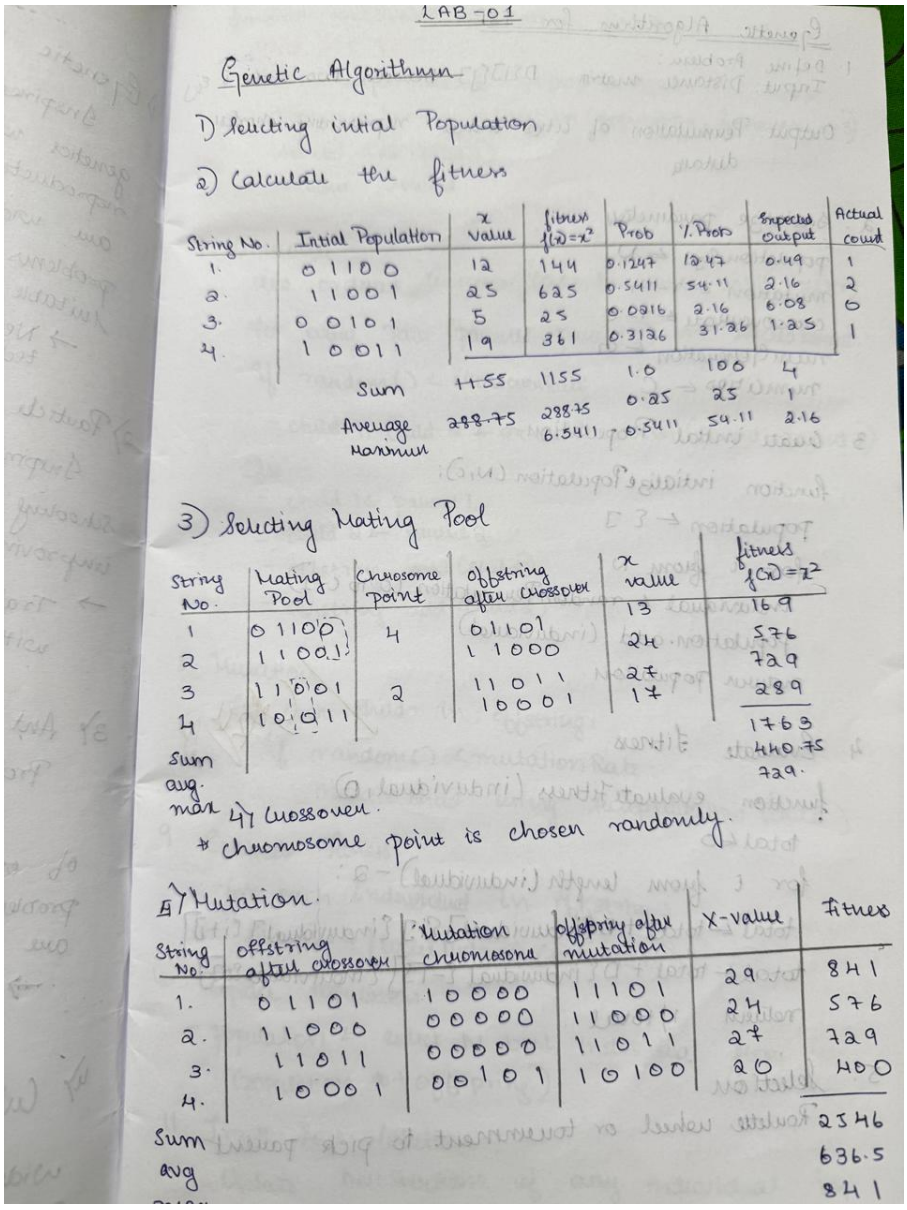
https://github.com/keertanabj/bislab

Program 1 : Genetic Algorithm

Problem statement:

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.

Algorithm:



LAB-01

Genetic Algorithm

1) Selecting initial Population

2) Calculate the fitness

String No.	Intial Population	x value	fitness $f(x)=x^2$	Prob	% Prob	Expected output	Actual count
1.	0 1 1 0 0	12	144	0.1247	12.47	0.49	1
2.	1 1 0 0 1	25	625	0.5411	54.11	2.16	2
3.	0 0 1 0 1	5	25	0.0216	2.16	0.08	0
4.	1 0 0 1 1	19	361	0.3126	31.26	1.25	1
	Sum	1155	1155	1.0	100	4	
	Average	288.75	288.75	0.25	25	1	
	Maximum		625	0.5411	54.11	2.16	

3) Selecting Mating Pool

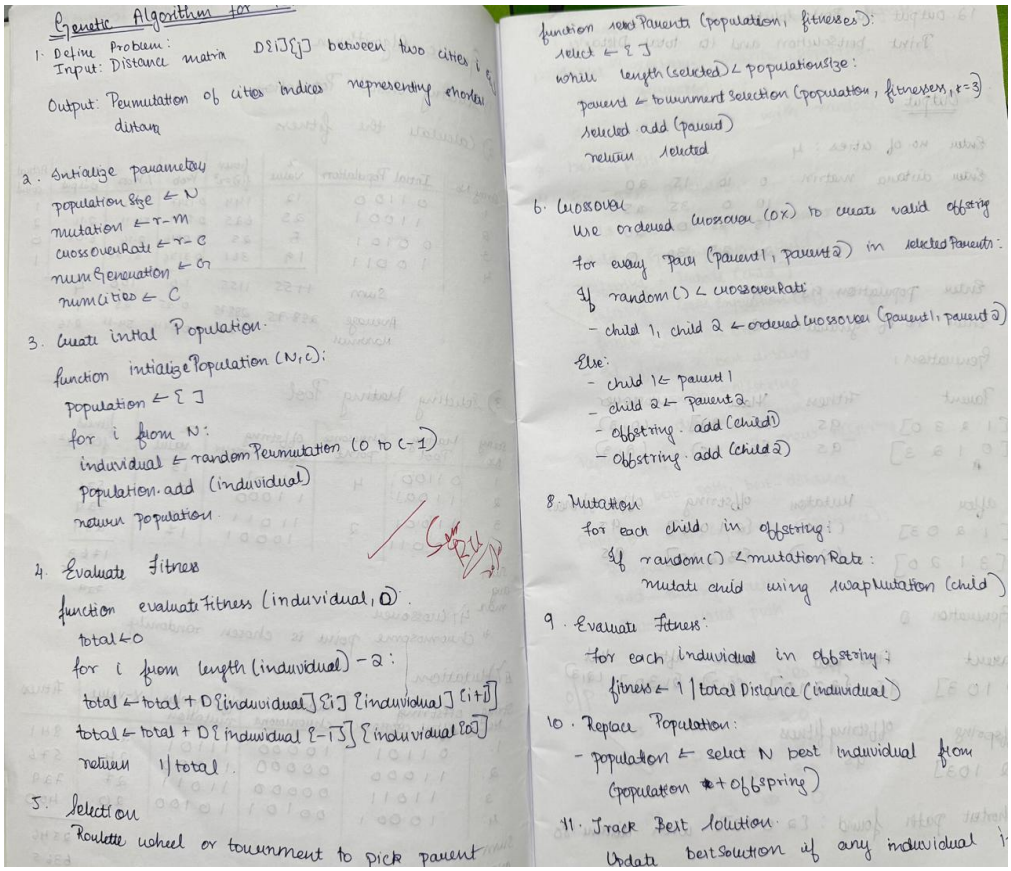
String No.	Mating Pool	Chromosome point	offstring after crossover	x value	fitness $f(x)=x^2$
1	0 1 1 0 0	4	0 1 1 0 1	13	169
2	1 1 0 0 1		1 1 0 0 0	24	576
3	1 1 0 0 1	2	1 1 0 1 1	27	729
4	1 0 0 1 1		1 0 0 0 1	17	289
Sum					1763
avg.					440.75
max					729.

4) Crossover.

* Chromosome point is chosen randomly.

5) Mutation.

String No	offstring after crossover	Mutation chromosome	offspring after mutation	X-value	fitness
1.	0 1 1 0 1	1 0 0 0 0	1 1 1 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
					841



Genetic Algorithm for

1. Define Problem:
Input: Distance matrix D[i][j] between two cities i
Output: Permutation of cities indices representing shortest distance

2. Initialize parameters
populationSize ← N
mutationRate ← r_m
crossoverRate ← r_c
numGenerations ← G
numCities ← C

3. Create intial Population.
function initializePopulation(N, C):
population ← [ ]
for i from N:
individual ← randomPermutation(0 to C-1)
population.add(individual)
return population

4. Evaluate fitness
function evaluateFitness(individual, D):
total ← 0
for i from length(individual) - 2:
total ← total + D[individual[i]][individual[i+1]]
total ← total + D[individual[-1]][individual[0]]
return 1/total

5. Selection
Roulette wheel or tournament to pick parent

function selectParents(population, fitnesses):
select ← [ ]
while length(selected) < populationSize:
parent ← tournamentSelection(population, fitnesses, t=3)
selected.add(parent)
return selected

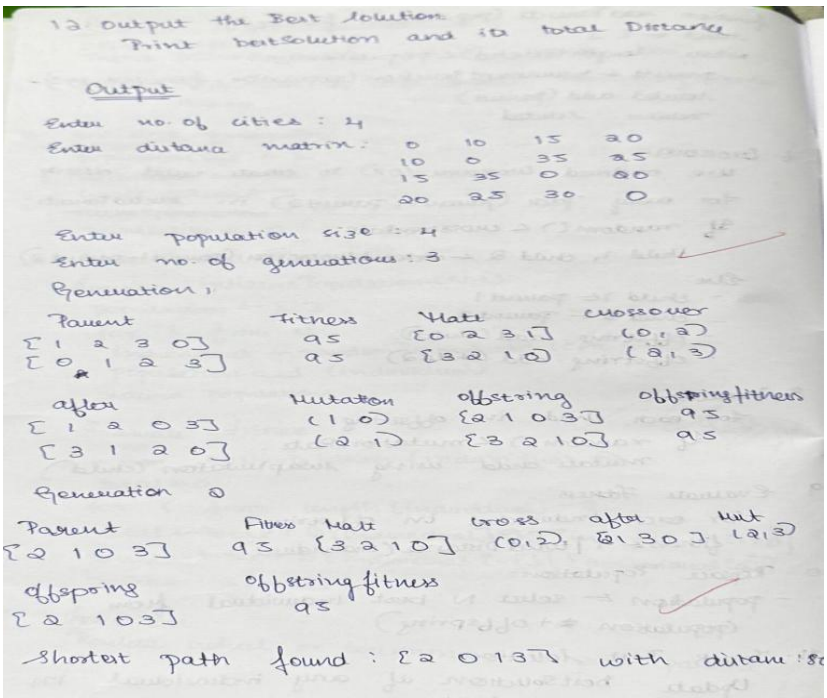
6. Crossover
Use ordered crossover (OX) to create valid offspring
for every pair (parent1, parent2) in selectedParents:
if random() < crossoverRate:
- child 1, child 2 ← orderedCrossover(parent1, parent2)
Else:
- child 1 ← parent 1
- child 2 ← parent 2
- offstring.add(child1)
- offstring.add(child2)

8. Mutation
for each child in offstring:
if random() < mutationRate:
mutate child using swapMutation(child)

9. Evaluate fitness:
for each individual in offstring:
fitness ← 1/totalDistance(individual)

10. Replace Population:
- population ← select N best individual from (population + offspring)

11. Track Best Solution.
Update bestSolution if any individual



12. Output the Best Solution
Print bestSolution and its total Distance

Output

Enter no. of cities : 4
Enter distance matrix:

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0

Enter population size : 4
Enter no. of generations : 3

Generation 1:

Parent	fitness	Mate	crossover
[1 2 3 0]	95	[0 2 3 1]	(0, 2)
[0 1 2 3]	95	[3 2 1 0]	(2, 3)

after	Mutation	offstring	offspring fitness
[1 2 0 3]	(1 0)	[2 1 0 3]	95
[3 1 2 0]	(2 1)	[3 2 1 0]	95

Generation 0

Parent	Fitness	Mate	cross	after	Mut
[2 1 0 3]	95	[3 2 1 0]	(0, 2)	[2 1 3 0]	(2, 3)

offspring	offstring fitness
[2 1 0 3]	95

Shortest path found : [2 0 1 3] with distance: 8

Code:

```
import random
def fitness(x):
    return x**2
def int_to_bin(x):
    return format(x, '05b')
def bin_to_int(b):
    return int(b, 2)
def tournament_selection(pop, k=3):
    selected = random.sample(pop, k)
    selected.sort(key=lambda x: fitness(x), reverse=True)
    return selected[0]
def crossover(p1, p2):
    b1, b2 = int_to_bin(p1), int_to_bin(p2)
    point = random.randint(1, 4)
    child1 = bin_to_int(b1[:point] + b2[point:])
    child2 = bin_to_int(b2[:point] + b1[point:])
    return child1, child2
def mutate(x, mutation_rate=0.1):
    if random.random() < mutation_rate:
        b = list(int_to_bin(x))
        pos = random.randint(0, 4)
        b[pos] = '1' if b[pos] == '0' else '0'
        return bin_to_int("".join(b))
    return x
def genetic_algorithm(initial_population=None, pop_size=6, generations=20, crossover_rate=0.8,
```

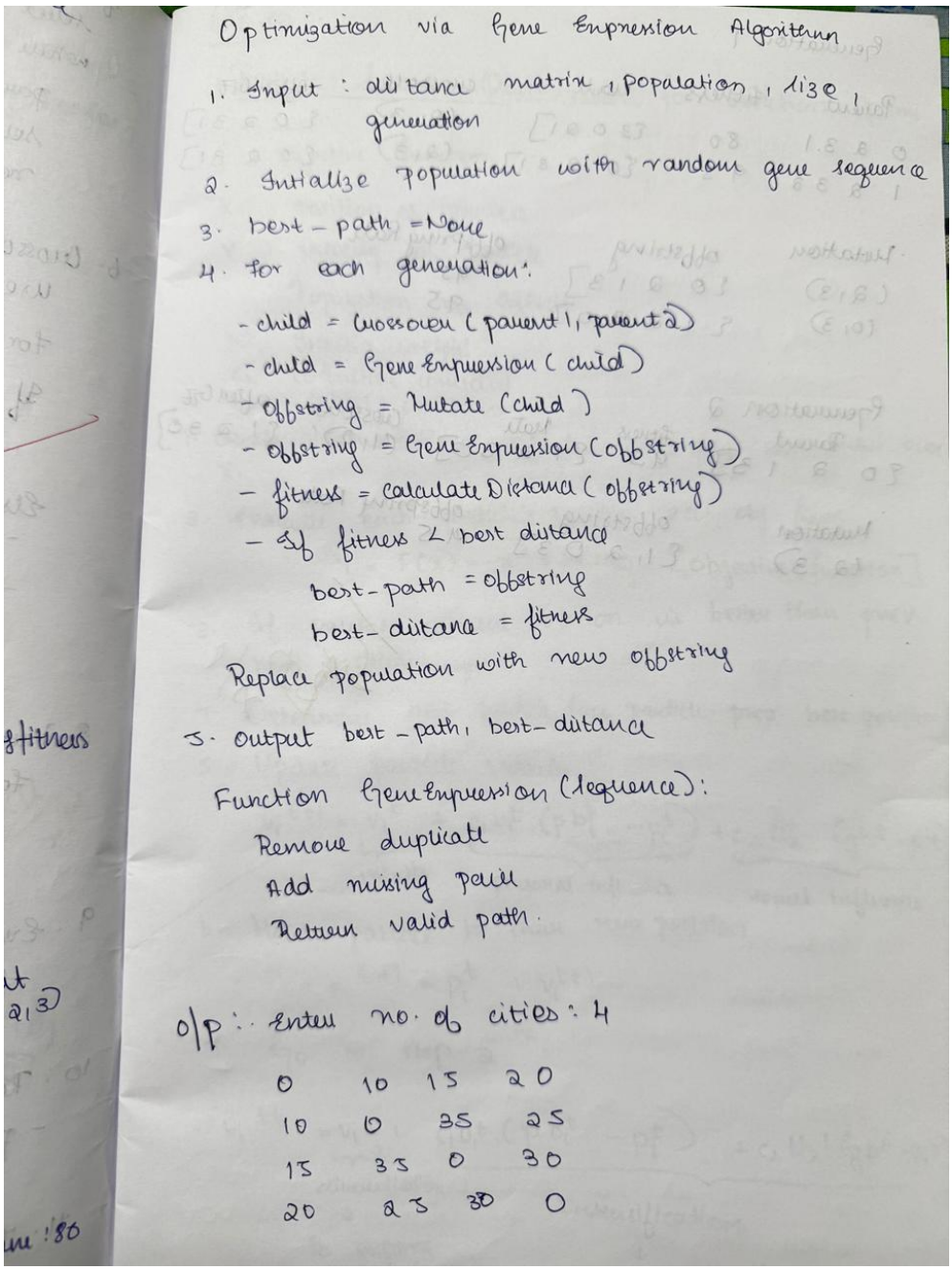
```
mutation_rate=0.1):
    if initial_population:
        population = initial_population[:pop_size]  # take only needed size
    else:
        population = [random.randint(0, 31) for _ in range(pop_size)]
    for gen in range(generations):
        population.sort(key=lambda x: fitness(x), reverse=True)
        best = population[0]
        print(f"Gen {gen}: Best x={best}, f(x)={fitness(best)}")
        new_pop = [best]
        while len(new_pop) < pop_size:
            parent1 = tournament_selection(population)
            parent2 = tournament_selection(population)
            if random.random() < crossover_rate:
                child1, child2 = crossover(parent1, parent2)
            else:
                child1, child2 = parent1, parent2
            child1 = mutate(child1, mutation_rate)
            child2 = mutate(child2, mutation_rate)
            new_pop.extend([child1, child2])
        population = new_pop[:pop_size]
    population.sort(key=lambda x: fitness(x), reverse=True)
    best = population[0]
    print(f"\nBest Solution: x={best}, f(x)={fitness(best)}")
custom_population = [3, 7, 15, 20, 25, 30]
genetic_algorithm(initial_population=custom_population, generations=5)
```

Program 2 : Optimization via Gene expression

Problem statement:

Gene Expression Algorithms (GEA) are inspired by the biological process of gene expression in living organisms. This process involves the translation of genetic information encoded in DNA into functional proteins. In GEA, solutions to optimization problems are encoded in a manner similar to genetic sequences. The algorithm evolves these solutions through selection, crossover, mutation, and gene expression to find optimal or near-optimal solutions. GEA is effective for solving complex optimization problems in various domains, including engineering, data analysis, and machine learning.

Algorithm:



Optimization via Gene Expression Algorithm

1. Input : distance matrix, population, size, generation
2. Initialize population with random gene sequence
3. best_path = None
4. for each generation:
   - child = Crossover (parent1, parent2)
   - child = Gene Expression (child)
   - offstring = Mutate (child)
   - offstring = Gene Expression (offstring)
   - fitness = calculate Distance (offstring)
   - If fitness < best distance
     best_path = offstring
     best_distance = fitness

   Replace population with new offstring
5. output best_path, best_distance

Function Gene Expression (sequence):
  Remove duplicate
  Add missing pair
  Return valid path.

o/p : Enter no. of cities : 4

0	10	15	20
10	0	35	25
15	35	0	30
20	25	30	0

Generation 1

Parent	fitness	Mate	crossover	After Crt
0 2 3 1	80	[3 0 2 1]	(0, 3)	[0 2 3 1]
1 2 3 2	95	[0 2 3 1]	(2, 3)	[0 2 3 1]

Mutation	offstring	offspring Rate
(2, 3)	[0 2 1 3]	95
(0, 3)	[1 2 3 0]	95

Generation 2

Parent	fitness	Mate	crossover	after Crt
[0 2 1 3]	95	[1 2 3 0]	(1, 2)	[1 2 3 0]

Mutation	offstring	offspring Rate
(2 3)	[1, 2 0 3]	95

Code:

```
import random
import math
cities = [
    (0, 0), (1, 5), (5, 2), (6, 6), (8, 3),
    (2, 1), (7, 7), (3, 3), (4, 4), (9, 0)
]
def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)
def total_distance(tour):
    dist = 0


    for i in range(len(tour)):
        city_a = cities[tour[i]]
        city_b = cities[tour[(i+1) % len(tour)]]
        dist += distance(city_a, city_b)
    return dist
def create_individual(n):
    gene = list(range(n))
    random.shuffle(gene)
    return gene
def mutate(individual, rate=0.1):
    ind = individual[:]
    for i in range(len(ind)):
        if random.random() < rate:
            j = random.randint(0, len(ind)-1)
            ind[i], ind[j] = ind[j], ind[i]
    return ind
def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted([random.randint(0, size-1) for _ in range(2)])
    child = [None]*size
    child[a:b+1] = parent1[a:b+1]
    p2_index = 0
    for i in range(size):
        if child[i] is None:
            while parent2[p2_index] in child:
                p2_index += 1
            child[i] = parent2[p2_index]
    return child
def genetic_algorithm(generations=100, pop_size=100, mutation_rate=0.1):
    num_cities = len(cities)
    population = [create_individual(num_cities) for _ in range(pop_size)]
    best = None
    best_dist = float('inf')
    for gen in range(generations):
        scored = [(ind, total_distance(ind)) for ind in population]
        scored.sort(key=lambda x: x[1])
        if scored[0][1] < best_dist:
```

```
            best = scored[0][0]
            best_dist = scored[0][1]
        new_pop = [best]
        while len(new_pop) < pop_size:
            p1 = random.choice(scored[:50])[0]
            p2 = random.choice(scored[:50])[0]
            child = crossover(p1, p2)
            child = mutate(child, mutation_rate)
            new_pop.append(child)
        population = new_pop
        if gen % 20 == 0:
            print(f"Gen {gen}: Best distance = {best_dist:.2f}")


    for i in range(len(tour)):
        city_a = cities[tour[i]]
        city_b = cities[tour[(i+1) % len(tour)]]
        dist += distance(city_a, city_b)
    return dist
def create_individual(n):
    gene = list(range(n))
    random.shuffle(gene)
    return gene
def mutate(individual, rate=0.1):
    ind = individual[:]
    for i in range(len(ind)):
        if random.random() < rate:
            j = random.randint(0, len(ind)-1)
            ind[i], ind[j] = ind[j], ind[i]
    return ind
def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted([random.randint(0, size-1) for _ in range(2)])
    child = [None]*size
    child[a:b+1] = parent1[a:b+1]
    p2_index = 0
    for i in range(size):
        if child[i] is None:
            while parent2[p2_index] in child:
                p2_index += 1
            child[i] = parent2[p2_index]
    return child
def genetic_algorithm(generations=100, pop_size=100, mutation_rate=0.1):
    num_cities = len(cities)
    population = [create_individual(num_cities) for _ in range(pop_size)]
    best = None
    best_dist = float('inf')
    for gen in range(generations):
        scored = [(ind, total_distance(ind)) for ind in population]
        scored.sort(key=lambda x: x[1])
            if scored[0][1] < best_dist: best = scored[0][0] best_dist = scored[0][1]
```

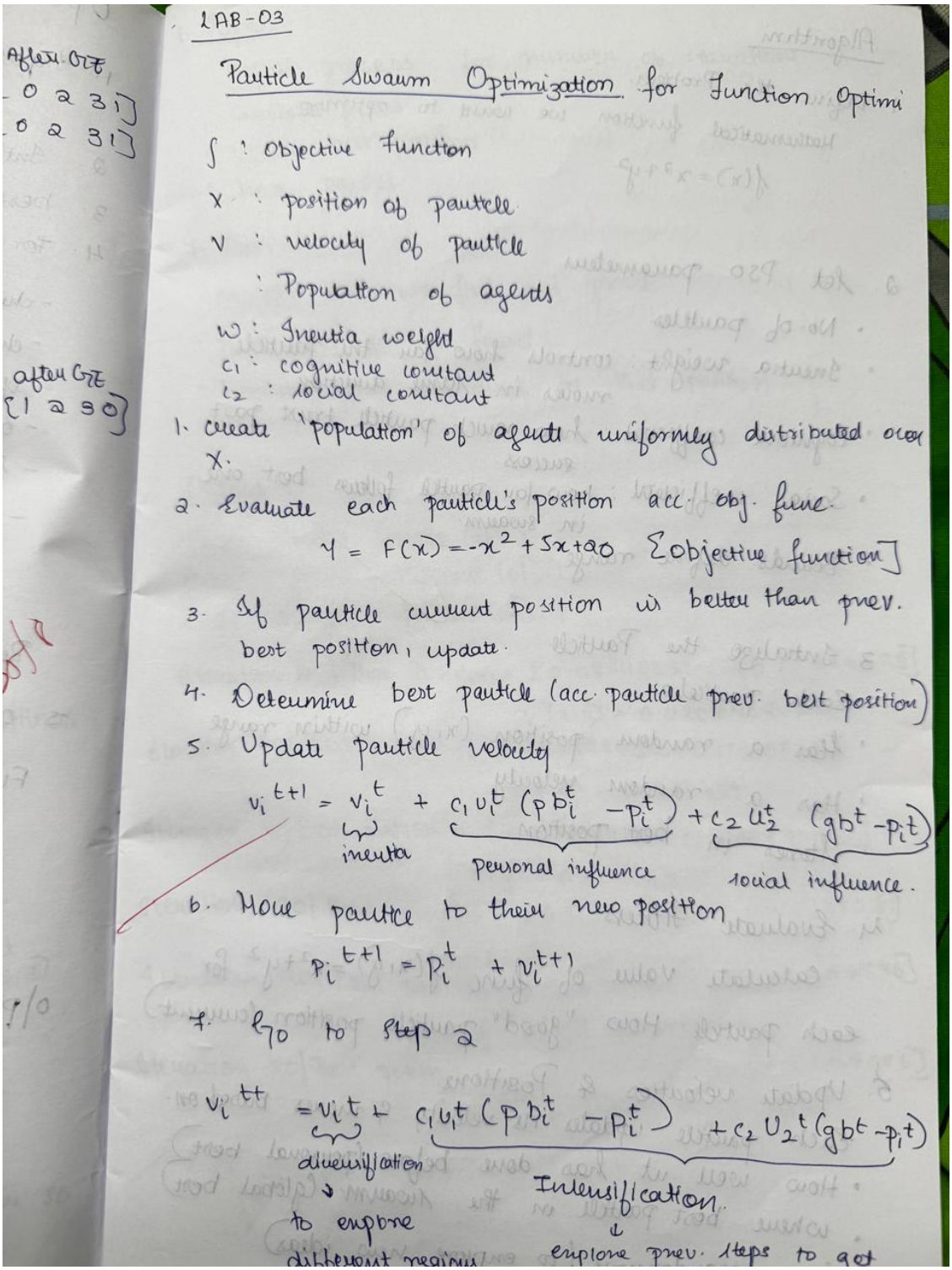
```
        return
        new_pop = [best]
        while len(new_pop) < pop_size:
            p1 = random.choice(scored[:50])[0]
            p2 = random.choice(scored[:50])[0]
            child = crossover(p1, p2)
            child = mutate(child, mutation_rate)
            new_pop.append(child)
        population = new_pop
        if gen % 20 == 0:
            print(f"Gen {gen}: Best distance = {best_dist:.2f}"
    best, best_dist
best_tour, best_dist = genetic_algorithm()
print("\nBest tour found:")
print(best_tour)
print(f"Total distance: {best_dist:.2f}")
```

Program 3 : Particle swarm Optimization

Problem statement:

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.

Algorithm:



LAB-03

Particle Swarm Optimization for Function Optimi

f : objective function

x : position of particle

v : velocity of particle

: Population of agents

w : Inertia weight

c_1 : cognitive constant

c_2 : social constant

1. create 'population' of agents uniformly distributed over x.
2. Evaluate each particle's position acc. obj. func.

$y = f(x) = -x^2 + 5x + 20$ [objective function]

3. If particle current position is better than prev. best position, update.
4. Determine best particle (acc. particle prev. best position)
5. Update particle velocity

$$v_i^{t+1} = v_i^t + c_1 u_1^t (pb_i^t - p_i^t) + c_2 u_2^t (gb^t - p_i^t)$$

v_i^t : inertia; $c_1 u_1^t (pb_i^t - p_i^t)$: personal influence; $c_2 u_2^t (gb^t - p_i^t)$: social influence.

6. Move particle to their new position

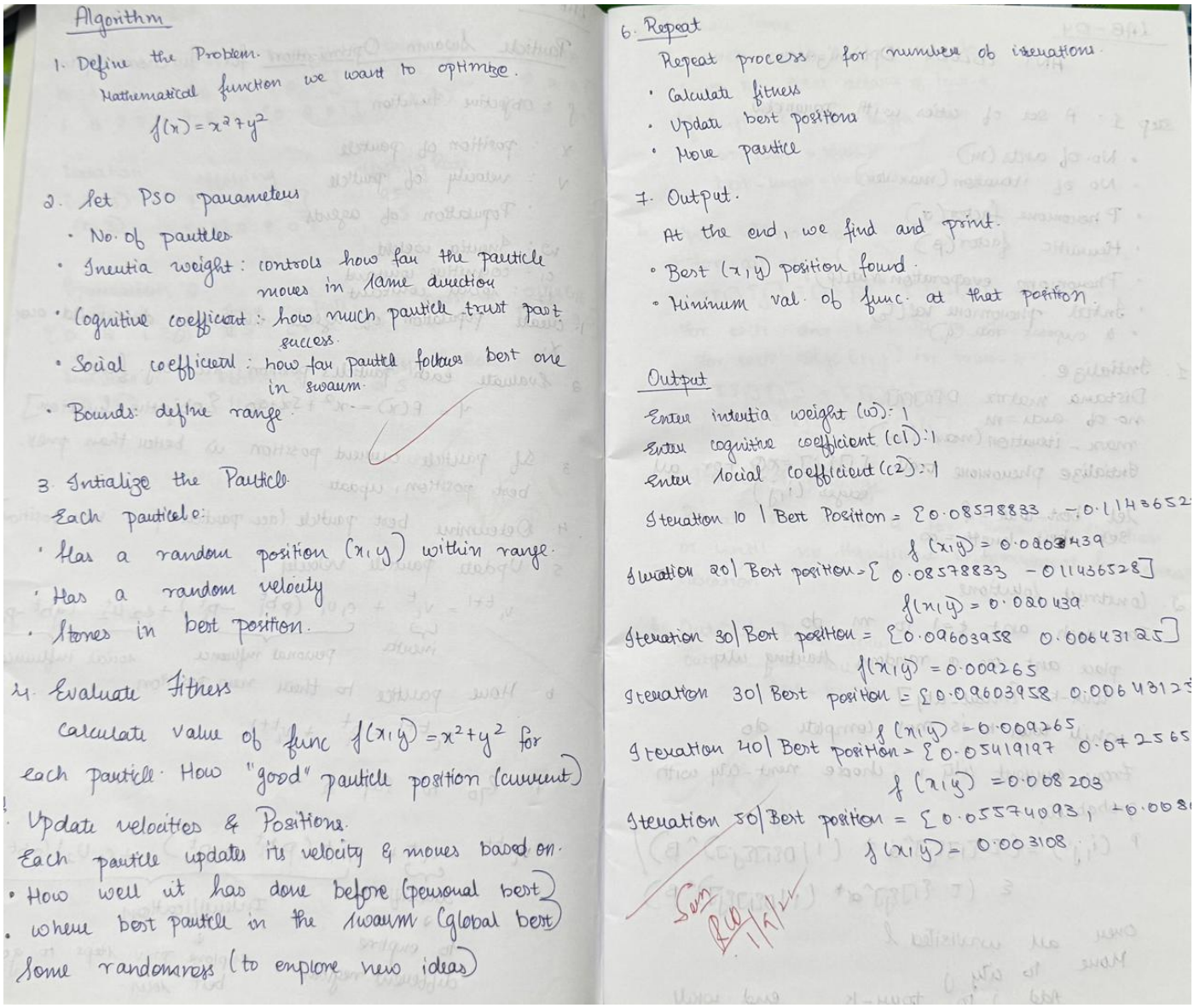
$$p_i^{t+1} = p_i^t + v_i^{t+1}$$

7. Go to step 2

$$v_i^{t+1} = v_i^t + c_1 u_1^t (pb_i^t - p_i^t) + c_2 u_2^t (gb^t - p_i^t)$$

v_i^t : diversification → to explore different regions

$c_1 u_1^t (pb_i^t - p_i^t) + c_2 u_2^t (gb^t - p_i^t)$: Intensification → explore prev. steps to get



Code:

```
import numpy as np
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([3, 5, 7, 9, 11])
def objective_function(theta):
    theta_0, theta_1 = theta
    predictions = theta_0 + theta_1 * x_data
    errors = y_data - predictions
    return np.sum(errors**2)
num_particles = 30
num_iterations = 10
w = 0.7
c1 = 1.5
c2 = 2.1
```

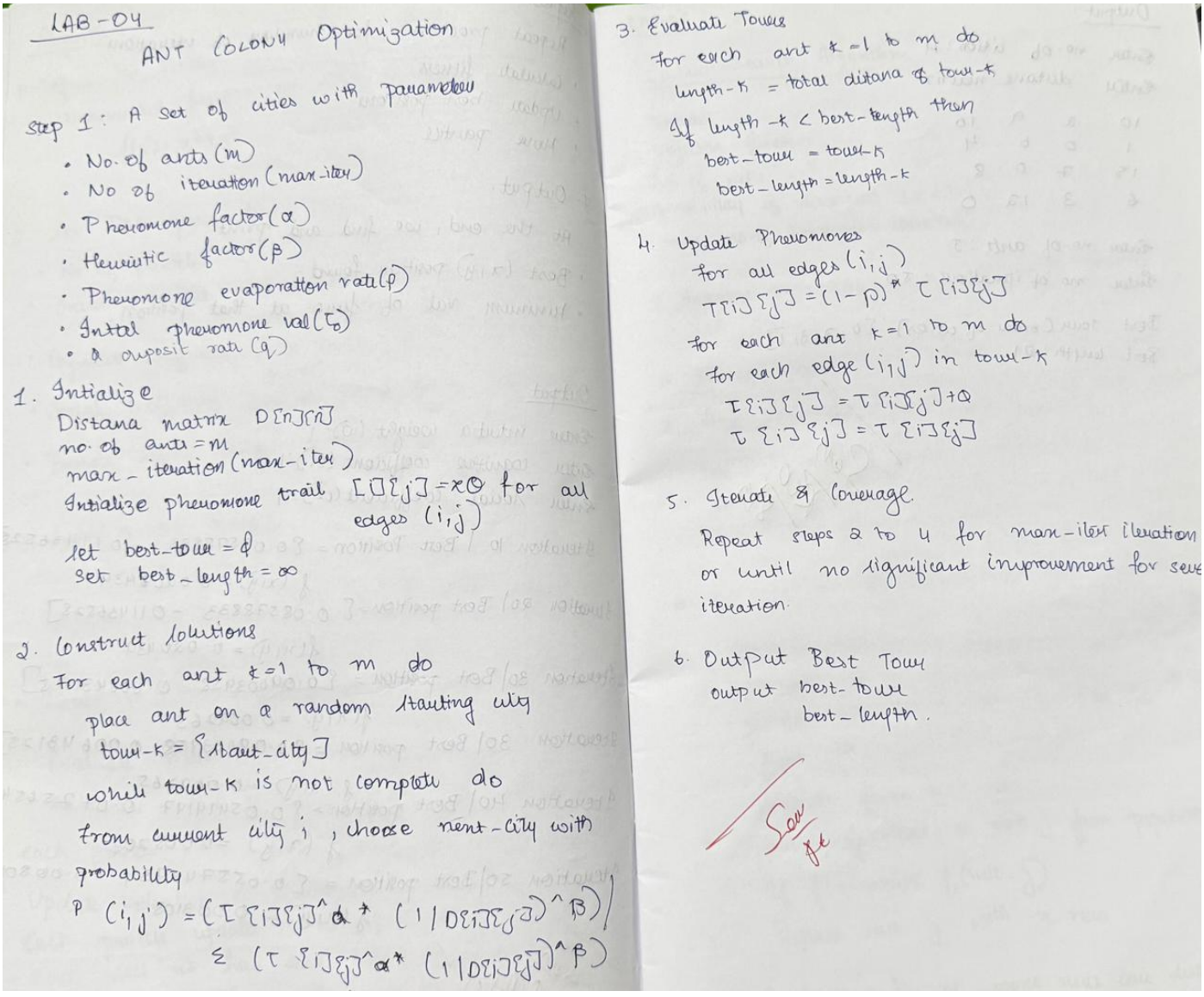
```
bounds = [(-10, 10), (-10, 10)]
positions = np.array([np.random.uniform(low, high, num_particles) for low, high in bounds]).T
velocities = np.random.uniform(-1, 1, (num_particles, 2))
personal_best_positions = np.copy(positions)
personal_best_values = np.array([objective_function(p) for p in personal_best_positions])
best_particle_index = np.argmin(personal_best_values)
global_best_position = personal_best_positions[best_particle_index]
global_best_value = personal_best_values[best_particle_index]
for iteration in range(num_iterations):
    for i in range(num_particles):
        fitness = objective_function(positions[i])
        if fitness < personal_best_values[i]:
            personal_best_values[i] = fitness
            personal_best_positions[i] = positions[i]
        if fitness < global_best_value:
            global_best_value = fitness
            global_best_position = positions[i]
    for i in range(num_particles):
        r1 = np.random.rand(2)
        r2 = np.random.rand(2)
        cognitive = c1 * r1 * (personal_best_positions[i] - positions[i])
        social = c2 * r2 * (global_best_position - positions[i])
        velocities[i] = w * velocities[i] + cognitive + social
        positions[i] += velocities[i]
        for dim in range(2):
            positions[i, dim] = np.clip(positions[i, dim], bounds[dim][0], bounds[dim][1])
    print(f"Iteration {iteration+1}/{num_iterations}, Best SSE: {global_best_value:.5f}")
print("\nBest parameters found:")
print("theta_0 =", global_best_position[0])
print("theta_1 =", global_best_position[1])
print("Minimum sum of squared errors:", global_best_value)
```

Program 4 : Ant Colony Optimization

Problem statement:

The foraging behavior of ants has inspired the development of optimization algorithms that can solve complex problems such as the Traveling Salesman Problem (TSP). Ant Colony Optimization (ACO) simulates the way ants find the shortest path between food sources and their nest. Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:



LAB-04

ANT COLONY Optimization

Step 1: A set of cities with parameters

- No. of ants (m)
- No of iteration (max-iter)
- Pheromone factor (α)
- Heuristic factor (β)
- Pheromone evaporation rate (ρ)
- Intial pheromone val (τ_0)
- A ouposit rate (q)

1. Intialize

Distana matrix D[n][n]
no. of ants = m
max_iteration (max-iter)
Intialize pheromone trail [i][j] = τ_0 for all edges (i, j)
let best_tour = ϕ
set best_length = ∞

2. Construct Solutions

For each ant k = 1 to m do
place ant on a random starting city
tour-k = [start_city]
while tour-k is not complete do
from current city i, choose next-city with probability

$$P(i,j) = \frac{(\tau[i][j]^{\alpha} * (1/D[i][j])^{\beta})}{\sum (\tau[i][j]^{\alpha} * (1/D[i][j])^{\beta})}$$

3. Evaluate Tours

for each ant k = 1 to m do
length-k = total distana of tour-k
If length-k < best-length then
best_tour = tour-k
best_length = length-k

4. Update Pheromones

for all edges (i, j)
$\tau[i][j] = (1-\rho)^{*} \tau[i][j]$
for each ant k = 1 to m do
for each edge (i, j) in tour-k
$\tau[i][j] = \tau[i][j] + Q$
$\tau[i][j] = \tau[i][j]$

5. Iterate & Converage.

Repeat steps 2 to 4 for max-iter iteration or until no significant improvement for seve iteration.

6. Output Best Tour

output best-tour
best_length.

Saw

Code:

```
import numpy as np
import random
NUM_CITIES = 10
NUM_ANTS = 20
NUM_ITERATIONS = 100
ALPHA = 1.0
BETA = 5.0
EVAPORATION = 0.5
Q = 100
np.random.seed(42)
cities = np.random.rand(NUM_CITIES, 2) * 100
dist_matrix = np.sqrt(((cities[:, np.newaxis, :] - cities[np.newaxis, :, :]) ** 2).sum(axis=2))
pheromone = np.ones((NUM_CITIES, NUM_CITIES))

best_distance = float('inf')
best_path = []
for iteration in range(NUM_ITERATIONS):
    all_paths = []
    all_distances = []
    for ant in range(NUM_ANTS):
        path = [random.randint(0, NUM_CITIES - 1)]
        while len(path) < NUM_CITIES:
            current_city = path[-1]
            probabilities = []
            for next_city in range(NUM_CITIES):
                if next_city not in path:
                    tau = pheromone[current_city][next_city] ** ALPHA
                    eta = (1 / dist_matrix[current_city][next_city]) ** BETA
                    probabilities.append(tau * eta)
                else:
                    probabilities.append(0)
            probabilities = np.array(probabilities)
            probabilities /= probabilities.sum()
            next_city = np.random.choice(range(NUM_CITIES), p=probabilities)
            path.append(next_city)
        path.append(path[0])  # Return to starting city
        distance = sum(dist_matrix[path[i]][path[i + 1]] for i in range(NUM_CITIES))
        all_paths.append(path)
        all_distances.append(distance)
        if distance < best_distance:
            best_distance = distance
            best_path = path
    pheromone *= (1 - EVAPORATION)
    for i in range(NUM_ANTS):
        for j in range(NUM_CITIES):
            from_city = all_paths[i][j]
            to_city = all_paths[i][j + 1]
            pheromone[from_city][to_city] += Q / all_distances[i]
```

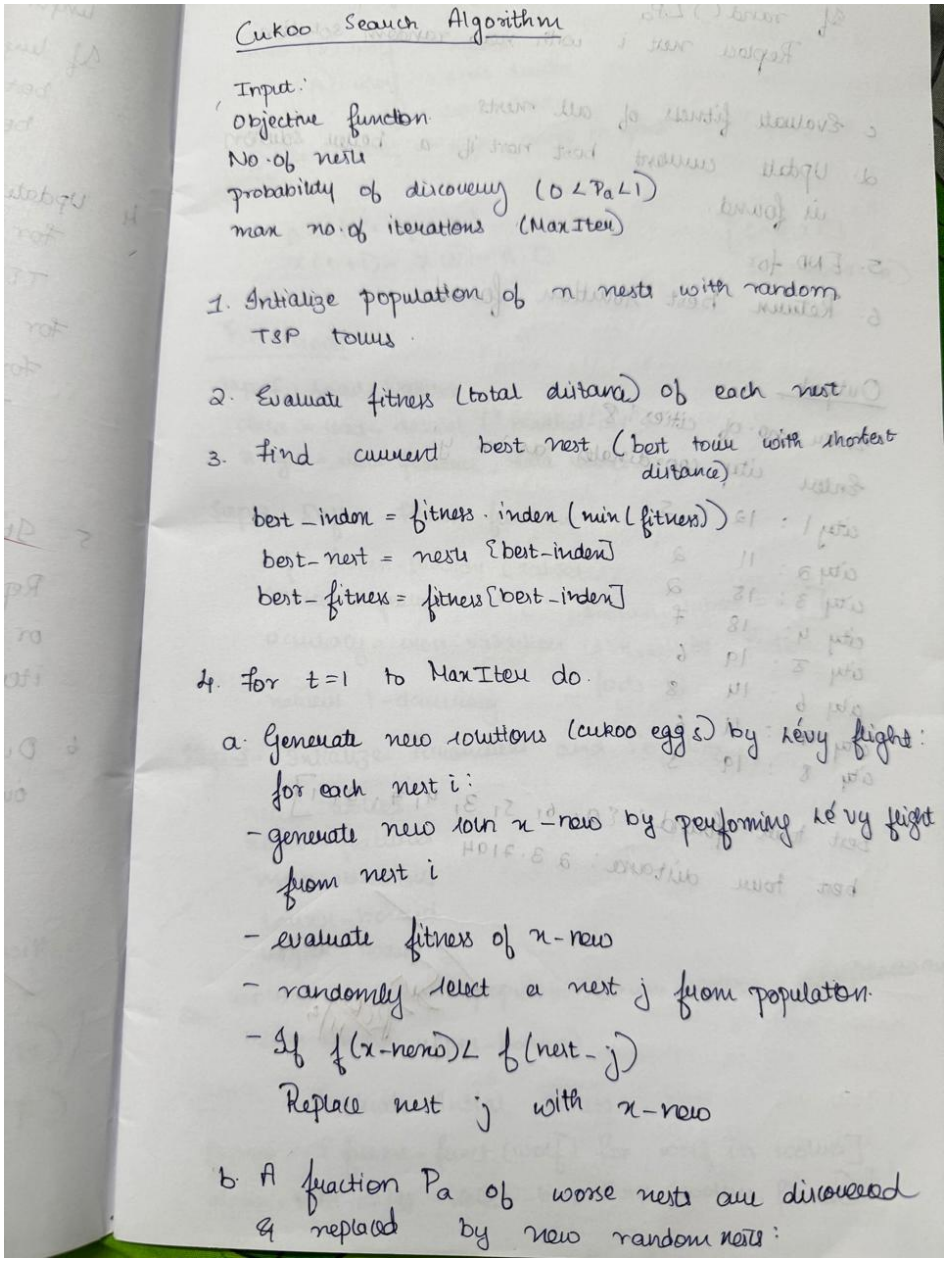
```
            pheromone[to_city][from_city] += Q / all_distances[i]
    if iteration % 10 == 0 or iteration == NUM_ITERATIONS - 1:
        print(f"Iteration {iteration}: Best Distance = {best_distance:.2f}")
print("\nBest Path Found:")
print(" -> ".join(map(str, best_path)))
print(f"Total Distance: {best_distance:.2f}")
```

Program 5 : Cuckoo search Optimization

Problem statement:

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:



Cukoo Search Algorithm

Input:
Objective function
No. of nests
probability of discovery ($0 < P_a < 1$)
max no. of iterations (MaxIter)

1. Initialize population of n nests with random TSP tours.
2. Evaluate fitness (total distance) of each nest
3. find current best nest (best tour with shortest distance)

best_index = fitness.index(min(fitness))
best_nest = nests[best_index]
best_fitness = fitness[best_index]

4. for t=1 to MaxIter do
   a. Generate new solutions (cukoo eggs) by Lévy flight:
      for each nest i:
      - generate new soln x_new by performing Lévy flight from nest i
      - evaluate fitness of x_new
      - randomly select a nest j from population.
      - If f(x_new) < f(nest_j)
        Replace nest j with x_new
   b. A fraction P_a of worse nests are discovered & replaced by new random nests:

- for each nest i
  If rand() < Pa
    Replace nest i with new random solution.

c. Evaluate fitness of all nests

d. Update current best nest if a better solution is found.

5. END for

6. Return best solution found.

Output

Enter no. of cities: 8

Enter city coordinates as 'x y'

city 1 : 12 5
city 2 : 11 2
city 3 : 15 2
city 4 : 18 4
city 5 : 19 6
city 6 : 14 8
city 7 : 11 6
city 8 : 19 5

Best tour found: [0, 6, 5, 3, 4, 7, 2, 1]

best tour distance: 23.7194

Sen
RCO
15/5

Code:

```
import random
import math
weights = [10, 20, 30, 40, 15, 25, 35]
values = [60, 100, 120, 240, 80, 150, 200]
capacity = 100  # Max weight capacity of the truck
n_items = len(weights)
n_nests = 15
max_iter = 50
pa = 0.25
def fitness(solution):
    total_weight = sum(w for w, s in zip(weights, solution) if s == 1)
    total_value = sum(v for v, s in zip(values, solution) if s == 1)
    if total_weight > capacity:
        return 0  # Penalize overweight solutions
    else:
        return total_value
def generate_nest():
    return [random.randint(0, 1) for _ in range(n_items)]
def levy_flight(Lambda=1.5):
    sigma_u = (math.gamma(1 + Lambda) * math.sin(math.pi * Lambda / 2) /
               (math.gamma((1 + Lambda) / 2) * Lambda * 2 ** ((Lambda - 1) / 2))) ** (1 / Lambda)

    u = random.gauss(0, sigma_u)
    v = random.gauss(0, 1)
    step = u / (abs(v) ** (1 / Lambda))
    return step
def get_cuckoo(nest, best_nest):
    new_nest = []
    for xi, bi in zip(nest, best_nest):
        step = levy_flight()
        val = xi + step * (xi - bi)
        s = 1 / (1 + math.exp(-val))
        new_val = 1 if s > 0.5 else 0
        new_nest.append(new_val)
    return new_nest
def cuckoo_search():
    nests = [generate_nest() for _ in range(n_nests)]
    fitness_values = [fitness(nest) for nest in nests]
    best_index = fitness_values.index(max(fitness_values))
    best_nest = nests[best_index][:]
    best_fitness = fitness_values[best_index]
    for _ in range(max_iter):
        for i in range(n_nests):
            new_nest = get_cuckoo(nests[i], best_nest)
            new_fitness = fitness(new_nest)
            if new_fitness > fitness_values[i]:
                nests[i] = new_nest
                fitness_values[i] = new_fitness
        for i in range(n_nests):
```

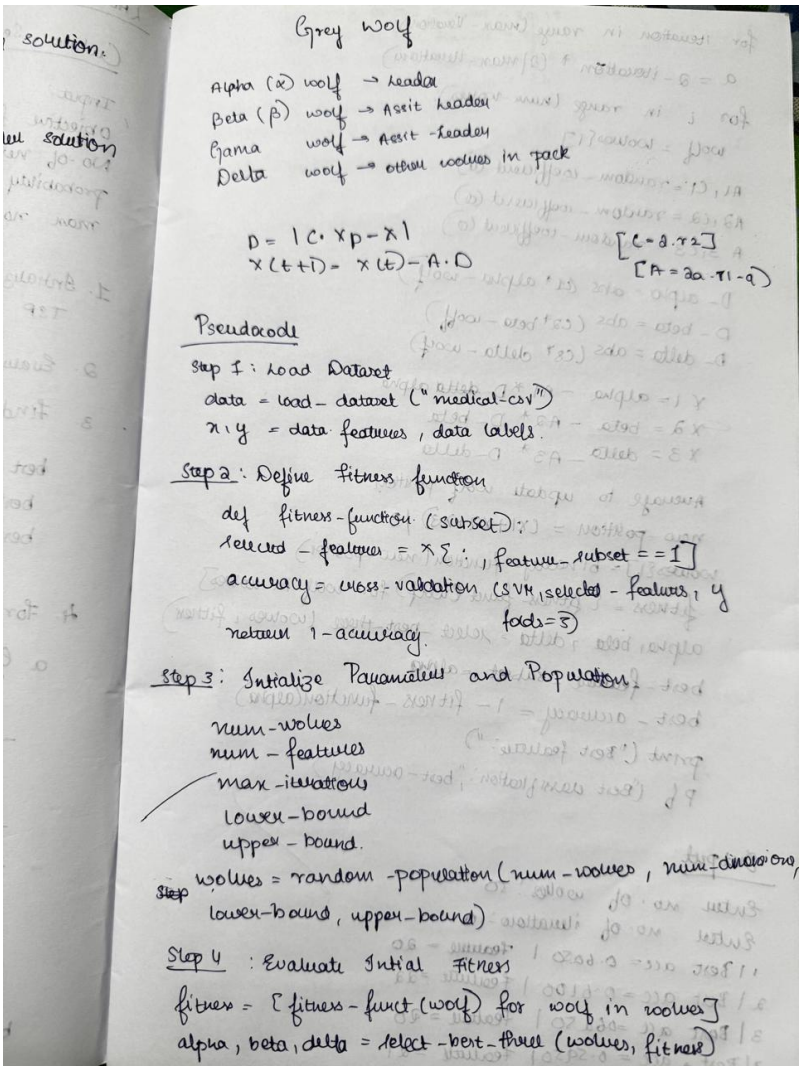
```
            if random.random() < pa:
                nests[i] = generate_nest()
                fitness_values[i] = fitness(nests[i])
        current_best_index = fitness_values.index(max(fitness_values))
        current_best_fitness = fitness_values[current_best_index]
        if current_best_fitness > best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_index][:]
    return best_nest, best_fitness
if __name__ == "__main__":
    best_solution, best_value = cuckoo_search()
    total_weight = sum(w for w, s in zip(weights, best_solution) if s == 1)
    print(f"Best packing solution (1 = selected): {best_solution}")
    print(f"Total value of supplies packed: {best_value}")
print(f"Total weight: {total_weight}")
```

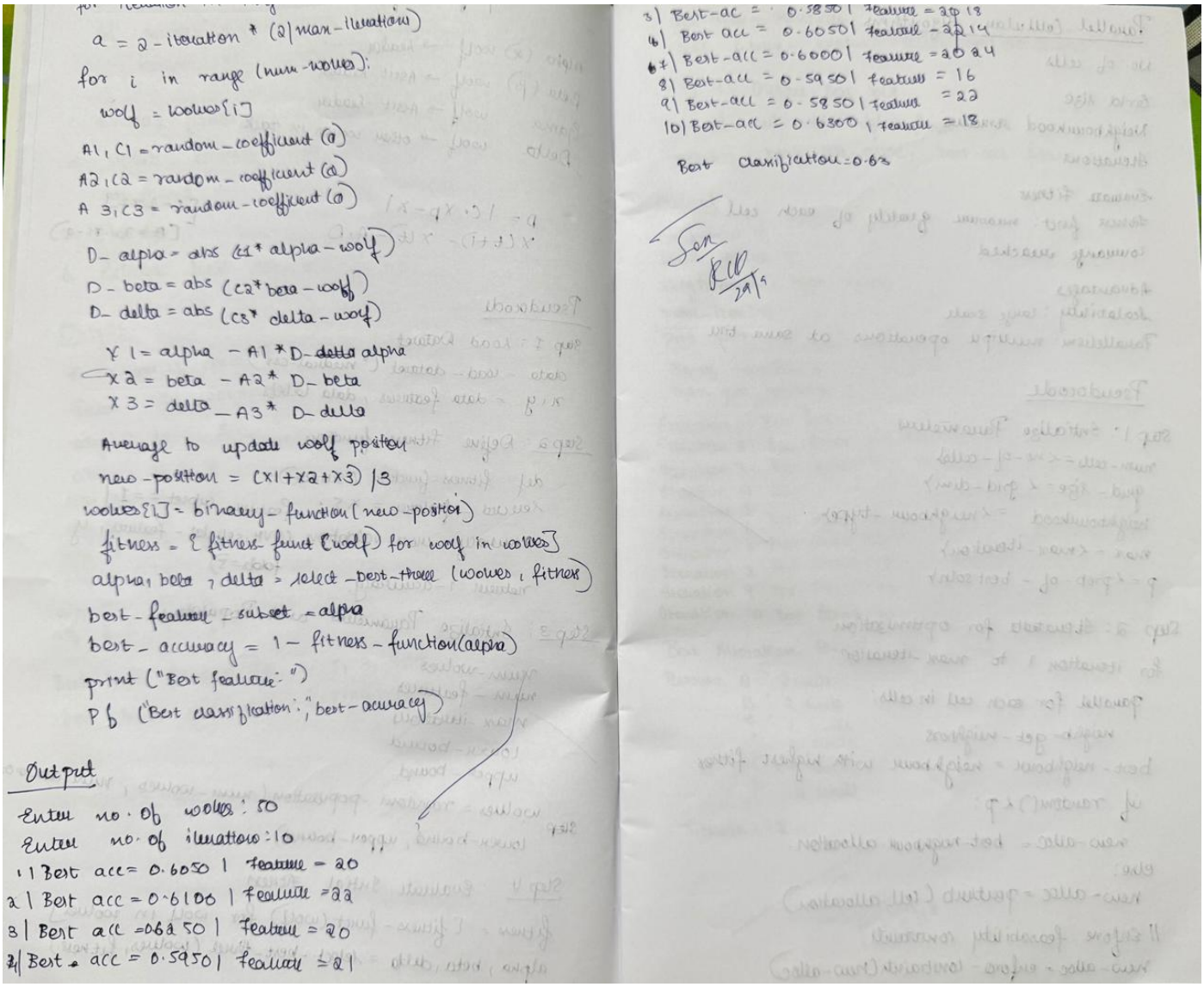
Program 6 : Grey Wolf Optimization

Problem statement:

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
def gwo(obj_func, dim, search_space, n_agents=20, max_iter=100):
    lb, ub = search_space
    wolves = np.random.uniform(lb, ub, (n_agents, dim))
    alpha, beta, delta = None, None, None
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")
    for t in range(max_iter):
        for i in range(n_agents):
            fitness = obj_func(wolves[i])
            if fitness < alpha_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = alpha_score, alpha
                alpha_score, alpha = fitness, wolves[i].copy()
```

```
            elif fitness < beta_score:
                delta_score, delta = beta_score, beta
                beta_score, beta = fitness, wolves[i].copy()
            elif fitness < delta_score:
                delta_score, delta = fitness, wolves[i].copy()
        a = 2 - 2 * (t / max_iter)
        for i in range(n_agents):
            for j in range(dim):
                r1, r2 = np.random.rand(), np.random.rand()

                A1, C1 = 2 * a * r1 - a, 2 * r2
                D_alpha = abs(C1 * alpha[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
                wolves[i][j] = np.clip((X1 + X2 + X3) / 3, lb, ub)
    return alpha, alpha_score
grid_size = (20, 20)
start, goal = np.array([0, 0]), np.array([19, 19])
obstacles = [
    (5, 5, 10, 10),
    (12, 0, 14, 14),
    (3, 15, 15, 17)
]
def is_collision(point):
    x, y = point.astype(int)
    if x < 0 or y < 0 or x >= grid_size[0] or y >= grid_size[1]:
        return True
    for ox1, oy1, ox2, oy2 in obstacles:
        if ox1 <= x <= ox2 and oy1 <= y <= oy2:
            return True
    return False
    waypoints = waypoints.reshape(-1, 2)
    path = [start] + [w.astype(int) for w in waypoints] + [goal]
    total_dist, penalty = 0, 0
    for i in range(len(path) - 1):
        dist = np.linalg.norm(path[i + 1] - path[i])
        total_dist += dist
        if is_collision(path[i + 1]):
            penalty += 100
    energy = 0
    for i in range(1, len(path) - 1):
        v1 = path[i] - path[i - 1]
        v2 = path[i + 1] - path[i]
```

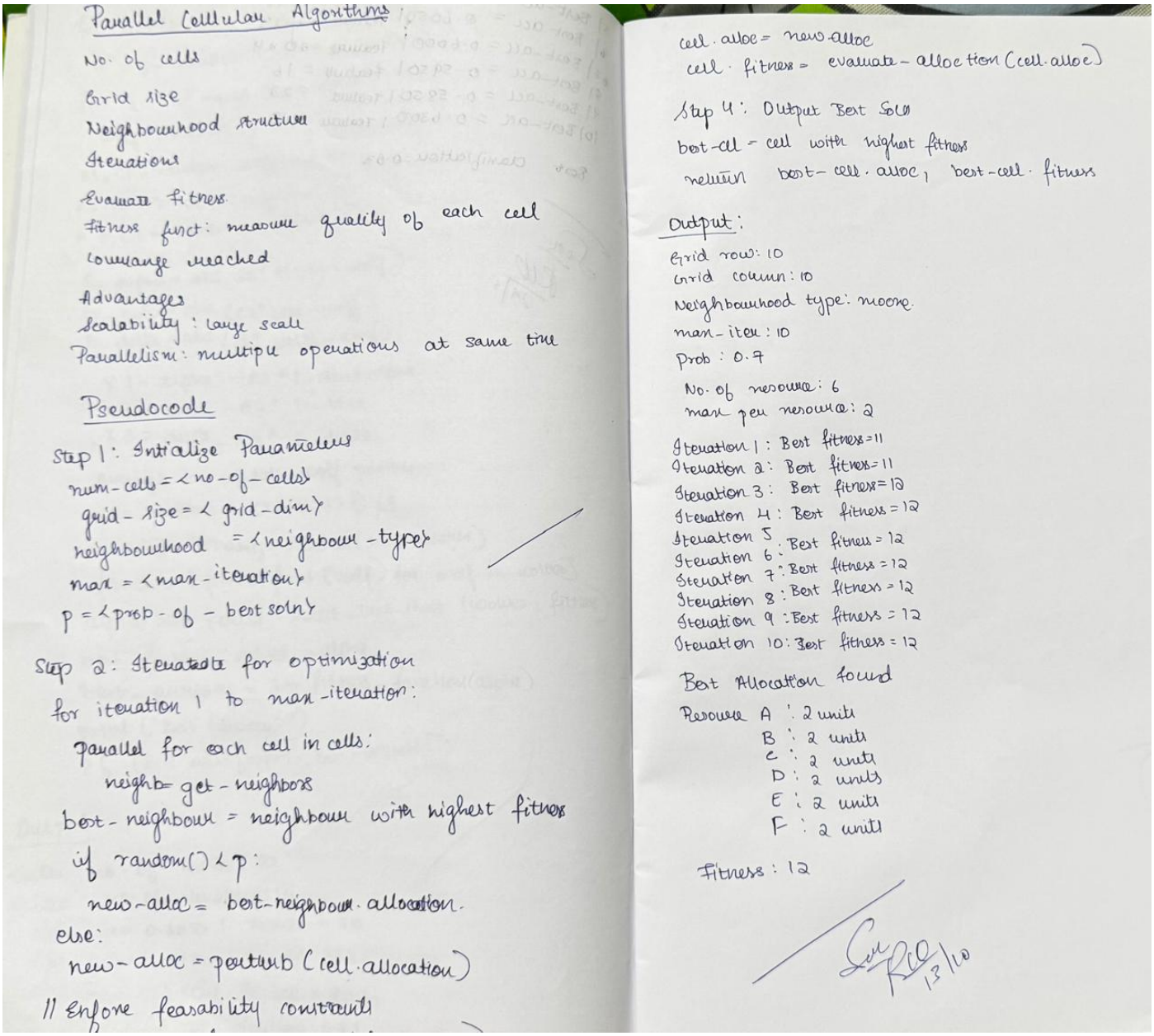
```
        if np.linalg.norm(v1) > 0 and np.linalg.norm(v2) > 0:
            cos_angle = np.dot(v1, v2) / (np.linalg.norm(v1) * np.linalg.norm(v2))
            angle = np.arccos(np.clip(cos_angle, -1, 1))
            energy += angle
    return total_dist + energy * 5 + penalty
n_waypoints = 5  # intermediate waypoints
dim = n_waypoints * 2
best_path, best_score = gwo(path_cost, dim, (0, grid_size[0]-1), n_agents=30, max_iter=200)
best_waypoints = best_path.reshape(-1, 2).astype(int)

final_path = np.vstack([start, best_waypoints, goal])
clean_path = []
for p in final_path:
    pt = tuple(map(int, p))
    if len(clean_path) == 0 or pt != clean_path[-1]:
        clean_path.append(pt)
print("Best Path Found:")
for p in clean_path:
    print(p)
print("\nPath Cost:", round(best_score, 2))
```

Program 7 : Parallel cellular Optimization

Problem statement:

Parallel Cellular Algorithms are inspired by the functioning of biological cells that operate in a highly parallel and distributed manner. These algorithms leverage the principles of cellular automata and parallel computing to solve complex optimization problems efficiently. Each cell represents a potential solution and interacts with its neighbors to update its state based on predefined rules. This interaction models the diffusion of information across the cellular grid, enabling the algorithm to explore the search space effectively. Parallel Cellular Algorithms are particularly suitable for large-scale optimization problems and can be implemented on parallel computing architectures for enhanced performance.

Algorithm:



Parallel Cellular Algorithms

No. of cells
Grid size
Neighbourhood structure
Iterations
Evaluate fitness
fitness funct: measure quality of each cell
coverage reached
Advantages
Scalability : Large scale
Parallelism: multiple operations at same time

Pseudocode

Step 1: Initialize Parameters
num_cells = <no_of_cells>
grid_size = <grid_dim>
neighbourhood = <neighbour_type>
max = <max_iteration>
p = <prob_of_best soln>

Step 2: Iterate for optimization
for iteration 1 to max_iteration:
parallel for each cell in cells:
neighb = get_neighbors
best_neighbour = neighbour with highest fitness
if random() < p:
new_alloc = best_neighbour.allocation
else:
new_alloc = perturb (cell.allocation)
// Enfore feasibility constraints
cell.alloc = new_alloc
cell.fitness = evaluate_allocation (cell.alloc)

Step 4: Output Best Soln
best_cell = cell with highest fitness
return best_cell.alloc, best_cell.fitness

Output:

Grid row: 10
Grid column: 10
Neighbourhood type: moore
max_iter: 10
Prob : 0.7
No. of resource: 6
max per resource: 2

Iteration 1: Best fitness = 11
Iteration 2: Best fitness = 11
Iteration 3: Best fitness = 12
Iteration 4: Best fitness = 12
Iteration 5: Best fitness = 12
Iteration 6: Best fitness = 12
Iteration 7: Best fitness = 12
Iteration 8: Best fitness = 12
Iteration 9: Best fitness = 12
Iteration 10: Best fitness = 12

Best Allocation found
Resource A : 2 units
B : 2 units
C : 2 units
D : 2 units
E : 2 units
F : 2 units

fitness : 12

13/10

Code:

```
import numpy as np
import random
from itertools import permutations
distance_matrix = np.array([
    [0, 2, 9, 10],
    [2, 0, 6, 4],
    [9, 6, 0, 8],
    [10, 4, 8, 0]
])
num_customers = distance_matrix.shape[0] - 1
population_size = 9
grid_dim = (3, 3)
num_vehicles = 2
def generate_individual():
    perm = list(range(1, num_customers + 1))
    random.shuffle(perm)
    return perm
population = [generate_individual() for _ in range(population_size)]
def fitness(individual):
    split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
    total_distance = 0
    for i in range(num_vehicles):

        route = [0] + individual[split_points[i]:split_points[i+1]] + [0]  # depot at start and end
        for j in range(len(route) - 1):
            total_distance += distance_matrix[route[j], route[j+1]]
    return total_distance
def get_neighbors(idx):
    r, c = divmod(idx, grid_dim[1])
    neighbors = []
    for dr in [-1, 0, 1]:
        for dc in [-1, 0, 1]:
            nr, nc = r + dr, c + dc
            if 0 <= nr < grid_dim[0] and 0 <= nc < grid_dim[1]:
                n_idx = nr * grid_dim[1] + nc
                if n_idx != idx:
                    neighbors.append(n_idx)
    return neighbors
def crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(random.sample(range(size), 2))
    child = [None] * size
    child[a:b] = parent1[a:b]
    pointer = b
    for gene in parent2[b:] + parent2[:b]:
        if gene not in child:
            if pointer == size:
                pointer = 0
```

```
            child[pointer] = gene
            pointer += 1
    return child
def mutate(individual):
    a, b = random.sample(range(len(individual)), 2)
    individual[a], individual[b] = individual[b], individual[a]
    return individual
def pca_iteration(pop):
    new_pop = pop.copy()
    for idx in range(len(pop)):
        neighbors = get_neighbors(idx)
        partner_idx = random.choice(neighbors)
        parent1 = pop[idx]
        parent2 = pop[partner_idx]
        child = crossover(parent1, parent2)
        if random.random() < 0.2:
            child = mutate(child)
        if fitness(child) < fitness(pop[idx]):
            new_pop[idx] = child
    return new_pop
num_generations = 25
for gen in range(num_generations):
    population = pca_iteration(population)
    best_fitness = min(fitness(ind) for ind in population)
    print(f"Generation {gen+1}: Best total distance = {best_fitness}")
best_individual = min(population, key=fitness)
print("\nBest route assignment (split evenly):")
split_points = np.linspace(0, num_customers, num_vehicles + 1, dtype=int)
for i in range(num_vehicles):
    route = [0] + best_individual[split_points[i]:split_points[i+1]] + [0]
    print(f"Vehicle {i+1} route: {route}")
print(f"Total distance: {fitness(best_individual)}")
```