VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Chirag S
USN:1BM23CS079

in partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug-2025 to Dec-2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Bio Inspired Systems (23CS5BSBIS)" carried out by **Chirag S (1BM23CS079),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements of the above mentioned subject and the work prescribed for the said degree.

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

Index

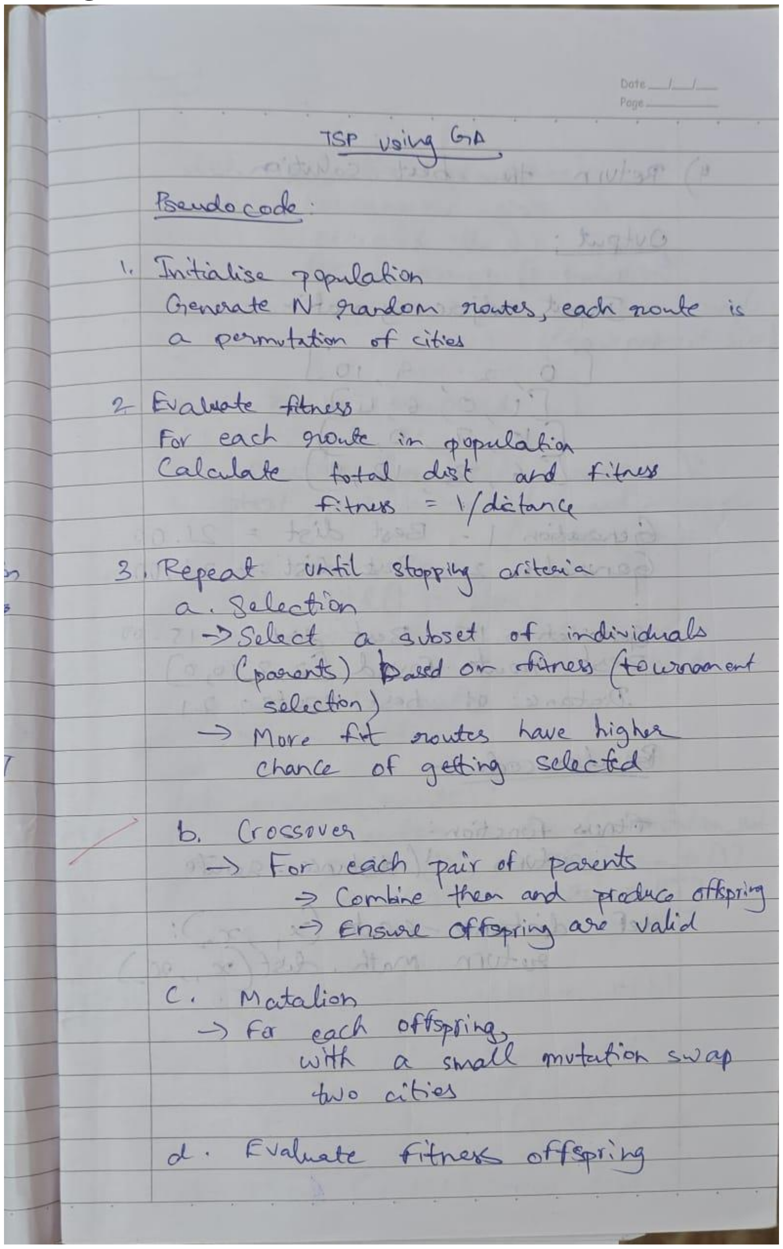
Sl. No.	Date	Experiment Title	Page No.
1	18/08/2025	Genetic Algorithm for Optimization Problems	4 - 7
2	25/08/2025	Optimization via Gene Expression Algorithms	8 - 10
3	01/09/2025	Particle Swarm Optimization for Function Optimization	11 – 13
4	08/09/2025	Ant Colony Optimization for the Traveling Salesman Problem	14 – 16
5	15/09/2025	Cuckoo Search (CS)	17 – 20
6	29/09/2025	Grey Wolf Optimizer (GWO)	21 – 22
7	13/10/2025	Parallel Cellular Algorithms and Programs	23 – 24

GitHub Link:
https://github.com/ChiragS00/1BM23CS079-BIS.git

<u>Program 1:</u> Genetic Algorithm for Optimization Problems

Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

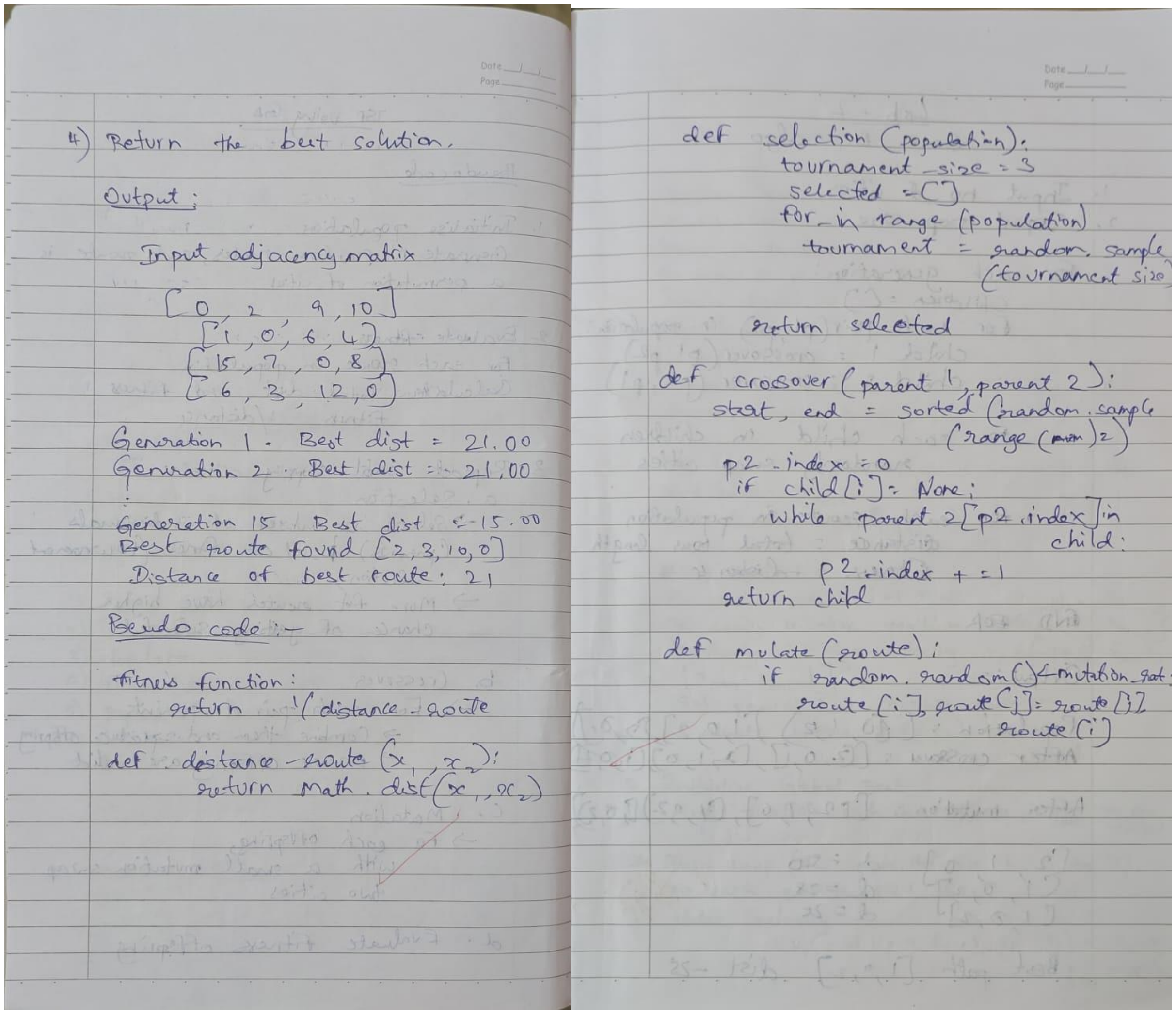


Date __/__/__
Page ____

TSP using GA

Pseudocode:

```
1. Initialise population
   Generate N random routes, each route is
   a permutation of cities

2. Evaluate fitness
   For each route in population
   Calculate total dist and fitness
           fitness = 1/distance

3. Repeat until stopping criteria
   a. Selection
      -> Select a subset of individuals
         (parents) based on fitness (tournament
         selection)
      -> More fit routes have higher
         chance of getting selected

   b. Crossover
      -> For each pair of parents
           -> Combine them and produce offspring
           -> Ensure offspring are valid

   c. Mutation
      -> For each offspring,
           with a small mutation swap
           two cities

   d. Evaluate fitness offspring
```



4) Return the best solution.

<u>Output</u>:

Input adjacency matrix

```
[0, 2, 9, 10]
 [1, 0, 6, 4]
 [15, 7, 0, 8]
 [6, 3, 12, 0]
```

```
Generation 1 . Best dist = 21.00
Generation 2 . Best dist = 21.00
:
Generation 15 . Best dist = 15.00
Best route found [2, 3, 10, 0]
 Distance of best route: 21
```

<u>Pseudo code</u>:-

```
Fitness function:
    return 1/distance_route

def distance_route(x1, x2):
        return Math.dist(x1, x2)

def selection(population):
        tournament_size = 3
        selected = []
        for_in range(population)
          tournament = random.sample
                         (tournament size)

    return selected

def crossover(parent 1, parent 2):
    start, end = sorted(random.sample
                    (range(num)2)
    p2_index = 0
    if child[i] = None;
        while parent 2[p2_index]in
                              child:
            p2_index + = 1
    return child

def mutate(route):
        if random.random()<mutation_rate:
            route[i], route[j] = route[j]
                                 route[i]
```

Code:

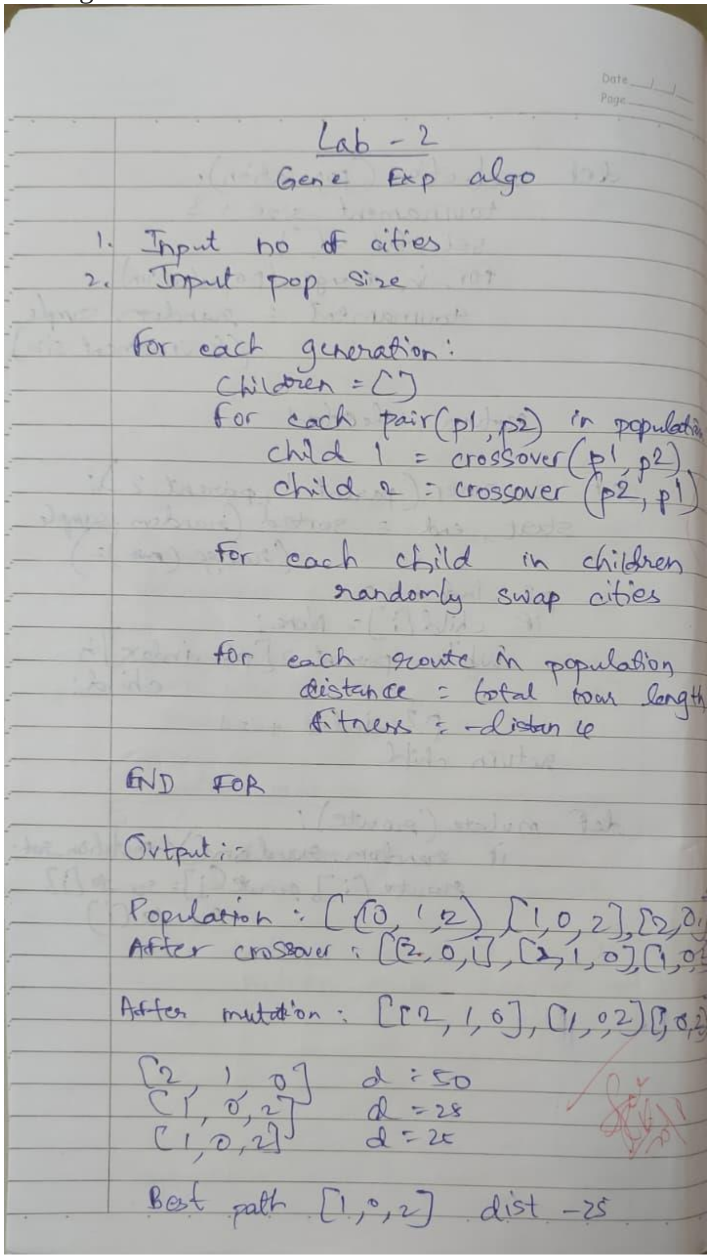
```
import random
import math

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

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

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

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

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

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

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

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

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

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

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

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

        population = new_population

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

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

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

genetic_algorithm()
```

<u>Program 2:</u> Optimization via Gene Expression Algorithms:

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

Algorithm:



Date __/__/__
Page ____

Lab - 2
Gene Exp algo

```
1. Input no of cities
2. Input pop size

For each generation:
    Children = []
    For each pair(p1, p2) in populatio
        child 1 = crossover(p1, p2)
        child 2 = crossover (p2, p1)

    For each child in children
        randomly swap cities

    For each route in population
        distance = total tour length
        fitness = -distance

END FOR
```

Output:-

Population : [(0, 1, 2) [1, 0, 2], [2, 0,
After crossover : [[2, 0, 1], [2, 1, 0], [1, 0,

After mutation : [[2, 1, 0], [1, 0, 2] [1, 0, 2

[2, 1, 0]   d = 50
[1, 0, 2]   d = 28
[1, 0, 2]   d = 25

Best path [1, 0, 2] dist -25

Code:

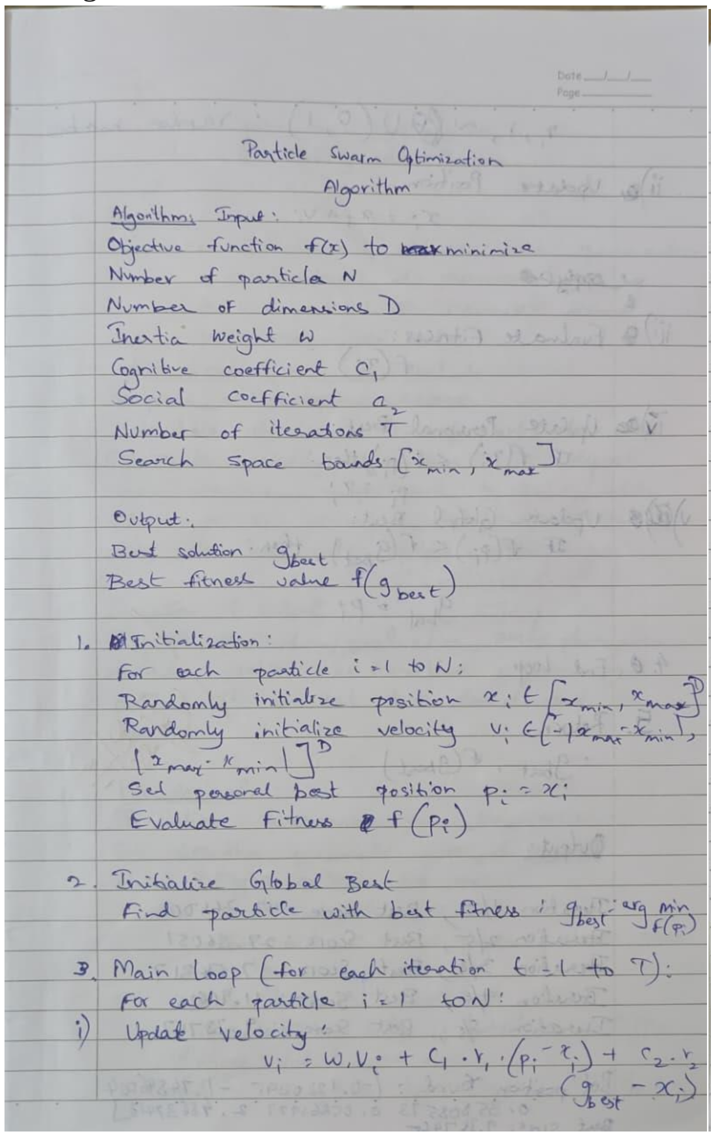
```
import random
import math

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

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

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

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

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

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

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

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

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

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

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

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

        population = new_population

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

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

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

genetic_algorithm()
```

Program 3: Particle Swarm Optimization for Function Optimization:

Particle Swarm Optimization (PSO) is inspired by the social behaviour of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

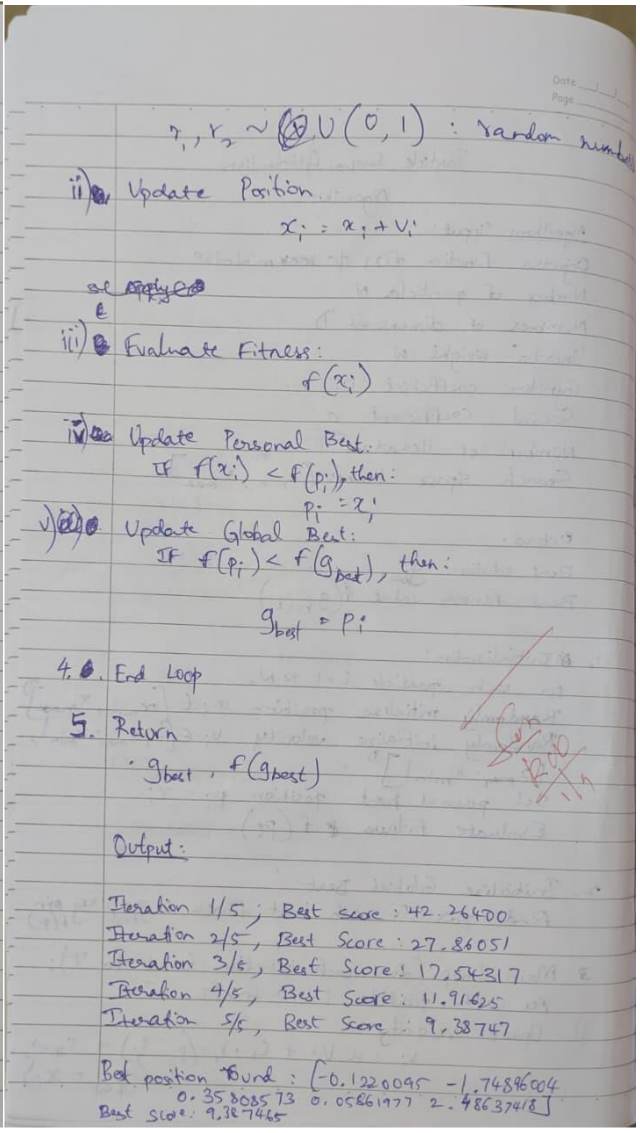


Particle Swarm Optimization
Algorithm

Algorithm: Input:
Objective function $f(x)$ to ~~max~~ minimize
Number of particle N
Number of dimensions D
Inertia weight w
Cognitive coefficient c_1
Social coefficient c_2
Number of iterations T
Search space bounds $[x_{min}, x_{max}]$

Output:
Best solution g_{best}
Best fitness value $f(g_{best})$

```
1. Initialization:
   For each particle i = 1 to N:
   Randomly initialize position $x_i \in [x_{min}, x_{max}]^D$
   Randomly initialize velocity $v_i \in [-|x_{max} - x_{min}|, |x_{max} - x_{min}|]^D$
   Set personal best position $p_i = x_i$
   Evaluate Fitness $f(p_i)$

2. Initialize Global Best
   Find particle with best fitness : $g_{best} = \arg\min f(p_i)$

3. Main Loop (for each iteration t = 1 to T):
   For each particle i = 1 to N:
   i) Update velocity:
        $v_i = w \cdot v_i + c_1 \cdot r_1 \cdot (p_i - x_i) + c_2 \cdot r_2 (g_{best} - x_i)$
```



```
        $r_1, r_2 \sim U(0,1)$ : random numbers
   ii) Update Position
        $x_i = x_i + v_i$

   iii) Evaluate Fitness:
        $f(x_i)$

   iv) Update Personal Best:
        IF $f(x_i) < f(p_i)$, then:
            $p_i = x_i$
   v) Update Global Best:
        IF $f(p_i) < f(g_{best})$, then:
            $g_{best} = p_i$

4. End Loop

5. Return
   $g_{best}$, $f(g_{best})$
```

Output:

```
Iteration 1/5 ; Best score : 42.26400
Iteration 2/5 , Best Score : 27.86051
Iteration 3/5 , Best Score : 17.54317
Iteration 4/5 , Best Score : 11.91625
Iteration 5/5 , Best Score : 9.38747

Best position found : [-0.1220095 -1.74896004
     0.35808573 0.05861977 2.48637418]
Best score: 9.387465
```

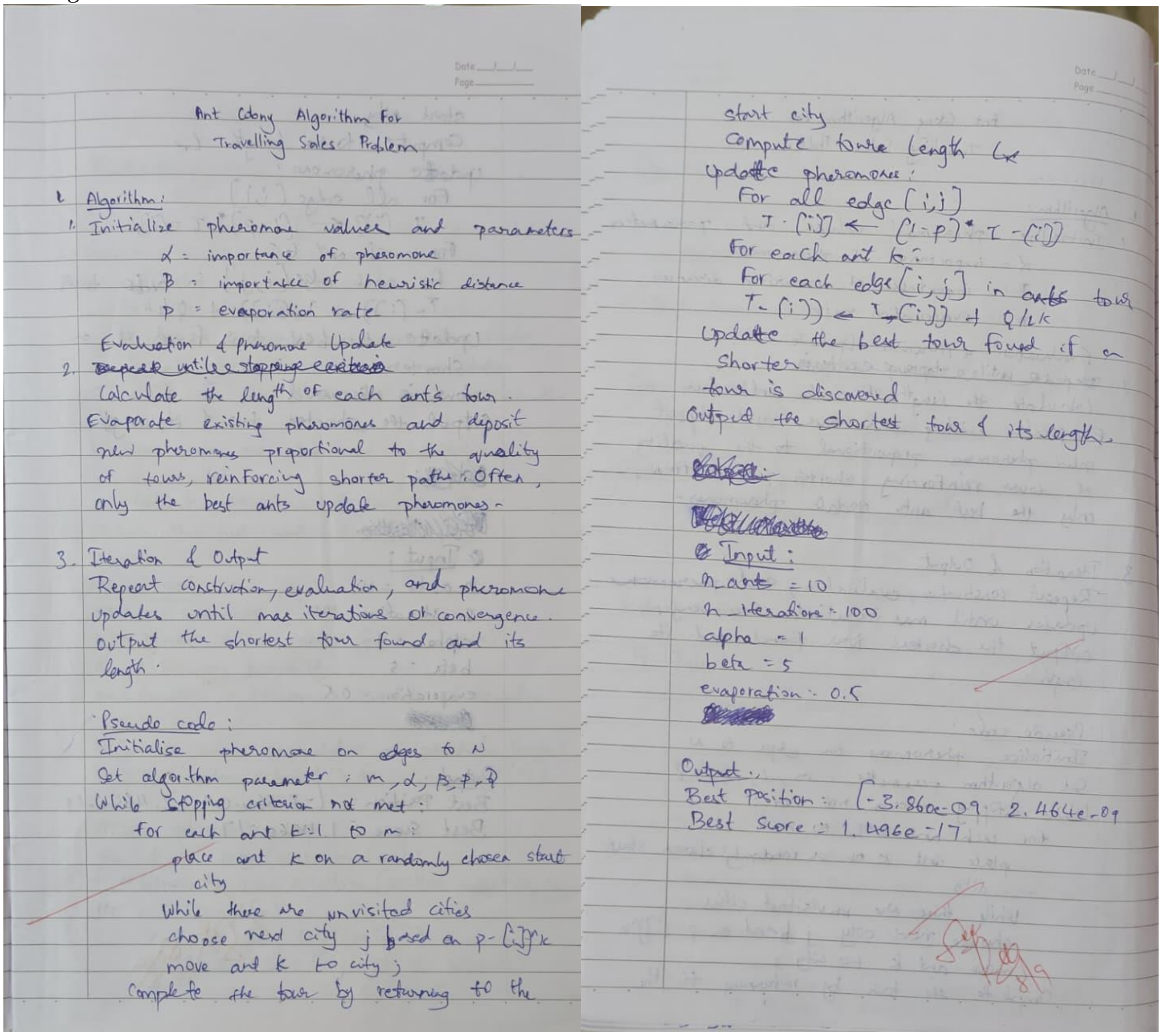
Code:

```
import random
import numpy as np

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

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

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

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

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

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

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

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

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

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

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

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

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

Program4: Ant Colony Optimization for the Traveling Salesman Problem:

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

Algorithm:



Date __/__/__
Page ____

Ant Colony Algorithm For
Travelling Sales Problem

Algorithm:

1. Initialize pheromone values and parameters
   α = importance of pheromone
   β = importance of heuristic distance
   p = evaporation rate

   Evaluation & Pheromone Update
2. ~~Repeat until a stopping criteria~~
   Calculate the length of each ant's tour.
   Evaporate existing pheromones and deposit new pheromones proportional to the quality of tours, reinforcing shorter paths. Often, only the best ants update pheromones.

3. Iteration & Output
   Repeat construction, evaluation, and pheromone updates until max iterations or convergence.
   output the shortest tour found and its length.

Pseudo code:

```
Initialise pheromone on edges to N
Set algorithm parameter : m, α, β, p, Q
While stopping criterion not met :
    for each ant k=1 to m :
        place ant k on a randomly chosen start
          city
        While there are unvisited cities
        choose next city j based on p-[i,j]^k
        move ant k to city j
    Complete the tour by returning to the
    start city
    Compute tour length Lk
  update pheromones :
    For all edge [i,j]
      τ-[i,j] ← [1-p] * τ-[i,j]
    For each ant k :
      For each edge [i,j] in ants tour
      τ-[i,j] ← τ-[i,j] + Q/Lk
  Update the best tour found if a
    shorter
  tour is discovered
Output the shortest tour & its length.
```

Date __/__/__
Page ____

[illegible]

[illegible]

Input:
n_ants = 10
n_iterations = 100
alpha = 1
beta = 5
evaporation = 0.5

Output:
Best position : [-3.860e-09 2.464e-09
Best score : 1.496e-17

[illegible]
8/9

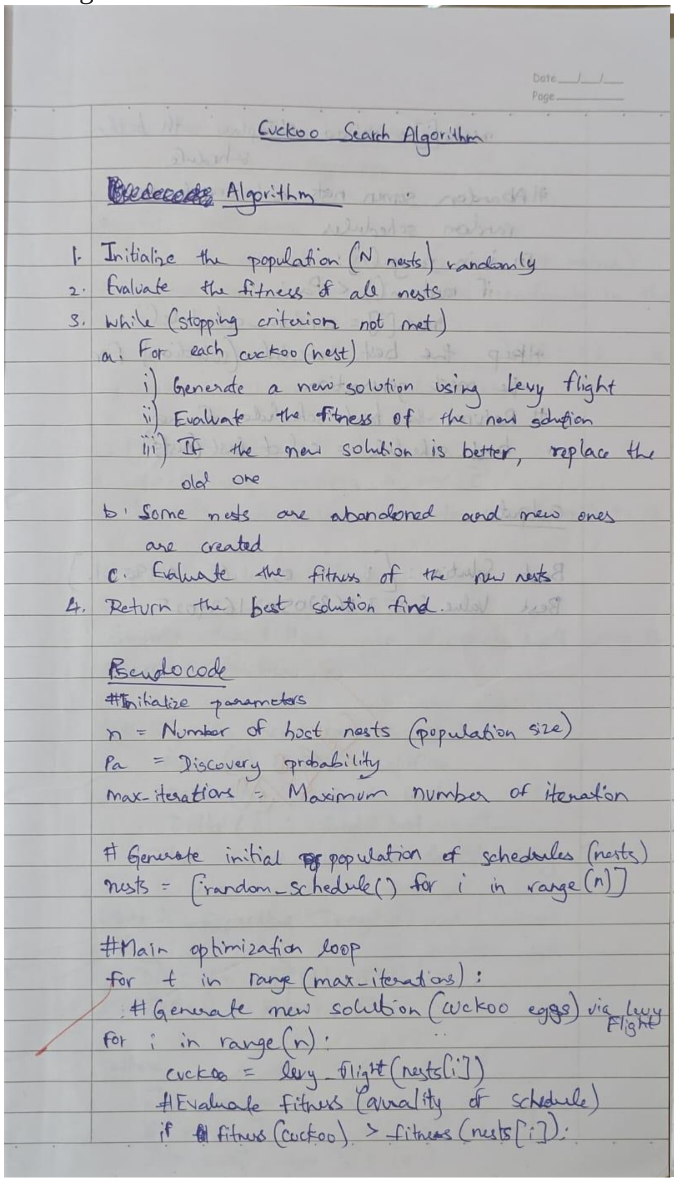
Code:

```
import numpy as np
import random


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


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


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

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

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


best_tour = None
best_length = float('inf')

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

    for _ in range(NUM_ANTS):
        tour = construct_tour()
        length = compute_tour_length(tour)
        all_tours.append((tour, length))

        if length < best_length:
            best_tour = tour
            best_length = length


    pheromone_matrix *= (1 - RHO)


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


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

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

Program 5: Cuckoo Search (CS):

Cuckoo Search (CS) is a nature-inspired optimization algorithm based on the brood parasitism of some cuckoo species. This behavior involves laying eggs in the nests of other birds, leading to the optimization of survival strategies. CS uses Lévy flights to generate new solutions, promoting global search capabilities and avoiding local minima. The algorithm is widely used for solving continuous optimization problems and has applications in various domains, including engineering design, machine learning, and data mining.

Algorithm:



Date __/__/__
Page ____

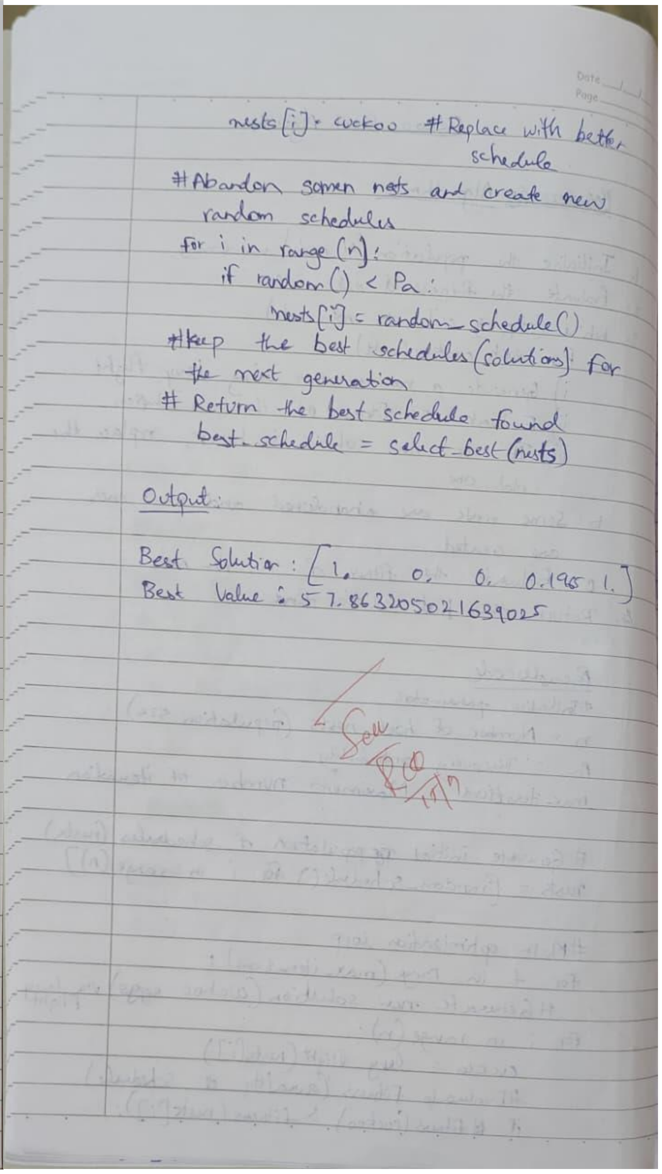
Cuckoo Search Algorithm

Algorithm

1. Initialize the population (N nests) randomly
2. Evaluate the fitness of all nests
3. While (stopping criterion not met)
   a. For each cuckoo (nest)
      i) Generate a new solution using Levy flight
      ii) Evaluate the fitness of the new solution
      iii) If the new solution is better, replace the old one
   b. Some nests are abandoned and new ones are created
   c. Evaluate the fitness of the new nests
4. Return the best solution find.

Pseudocode

```
#Initialize parameters
n = Number of host nests (population size)
Pa = Discovery probability
max-iterations = Maximum number of iteration

# Generate initial ~~pop~~ population of schedules (nests)
nests = [random_schedule() for i in range(n)]

#Main optimization loop
for t in range(max_iterations):
    # Generate new solution (cuckoo eggs) via levy Flight
for i in range(n):
        cuckoo = levy_flight(nests[i])
        #Evaluate fitness (quality of schedule)
        if fitness(cuckoo) > fitness(nests[i]):
```



Date __/__/__
Page ____

```
            nests[i] = cuckoo  # Replace with better
                                 schedule
    #Abandon somen nests and create new
      random schedules
    for i in range(n):
        if random() < Pa:
            nests[i] = random_schedule()
    #Keep the best schedules (solutions) for
      the next generation
    # Return the best schedule found
      best_schedule = select_best(nests)
```

Output:

```
Best Solution : [1.   0.   0.   0.195  1.]
Best Value : 57.86320502163902
```

[illegible]

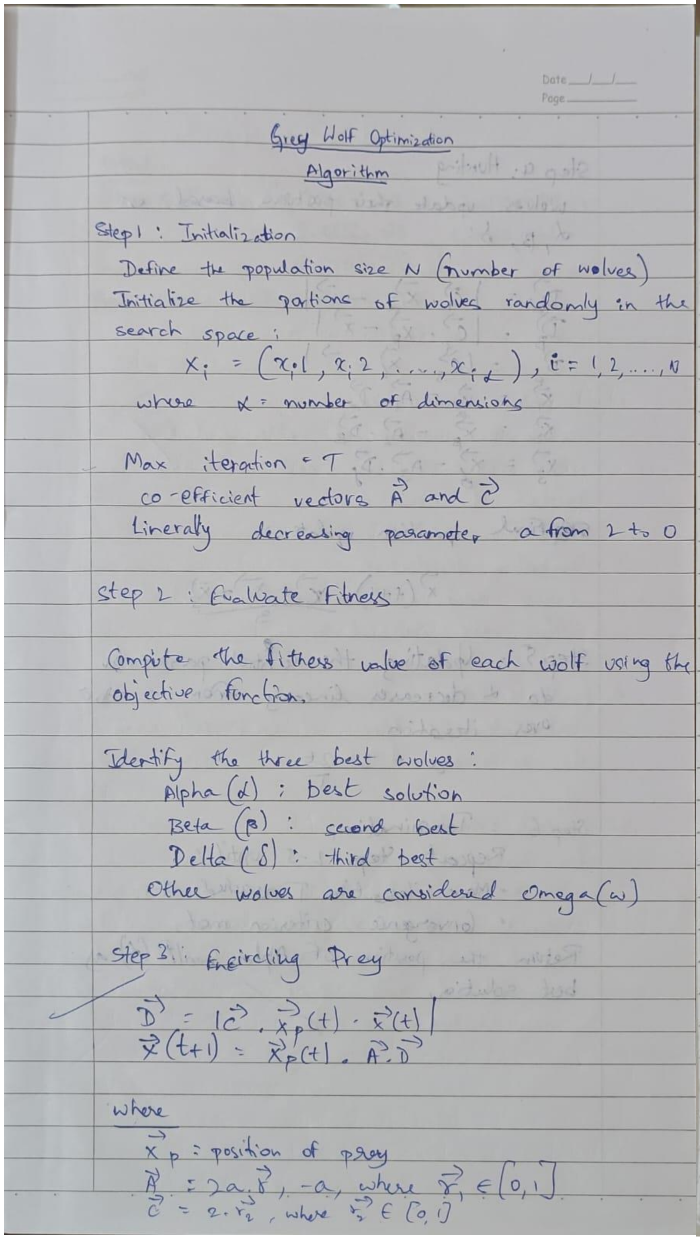
Code:

```
import numpy as np
import math

# --- Levy flight ---
def levy_flight(Lambda, dim):
    sigma = (math.gamma(1 + Lambda) * np.sin(np.pi * Lambda / 2) /
             (math.gamma((1 + Lambda) / 2) * Lambda * 2**((Lambda - 1) / 2)))**(1 / Lambda)
    u = np.random.normal(0, sigma, size=dim)
    v = np.random.normal(0, 1, size=dim)
    step = u / abs(v)**(1 / Lambda)
    return step

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

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

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

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

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

            # If new solution is better, replace
            if new_fitness > fitness[i]:
                nests[i] = new_nest
                nests_bin[i] = new_bin
                fitness[i] = new_fitness

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

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

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

        t += 1

    return best_bin, best_fitness

if _name___== "_main_":
    print("Enter the number of items:")
    n_items = int(input())

    weights = []
    values = []

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

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

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

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

    print("Enter abandonment probability Pa (default 0.25):")
    Pa = input()
    Pa = float(Pa) if Pa.strip() else 0.25

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

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

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

Program 6: Grey Wolf Optimizer (GWO):

The Grey Wolf Optimizer (GWO) algorithm is a swarm intelligence algorithm inspired by the social hierarchy and hunting behavior of grey wolves. It mimics the leadership structure of alpha, beta, delta, and omega wolves and their collaborative hunting strategies. The GWO algorithm uses these social hierarchies to model the optimization process, where the alpha wolves guide the search process while beta and delta wolves assist in refining the search direction. This algorithm is effective for continuous optimization problems and has applications in engineering, data analysis, and machine learning.

Algorithm:



Date __/__/__
Page ____

Grey Wolf Optimization

Algorithm

Step 1 : Initialization

Define the population size N (number of wolves)
Initialize the portions of wolves randomly in the search space :

$$X_i = (x_{i1}, x_{i2}, \ldots, x_{i\alpha}), \quad i = 1, 2, \ldots, N$$

where α = number of dimensions

Max iteration = T
co-efficient vectors $\vec{A}$ and $\vec{C}$
Lineraly decreasing parameter a from 2 to 0

step 2 : Evaluate Fitness

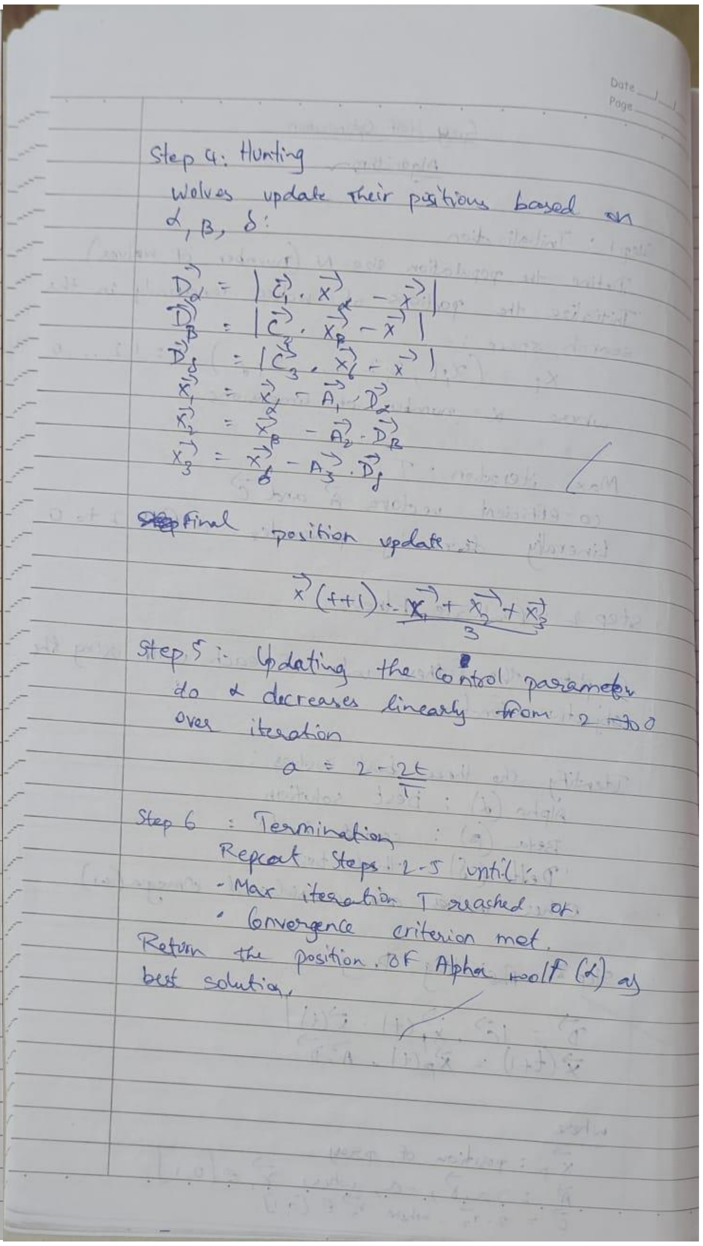
Compute the fitness value of each wolf using the objective function.

Identify the three best wolves :
Alpha (α) : best solution
Beta (β) : second best
Delta (δ) : third best
Other wolves are considered Omega (ω)

Step 3: Encircling Prey

$$\vec{D} = |\vec{C} \cdot \vec{X}_p(t) - \vec{x}(t)|$$
$$\vec{X}(t+1) = \vec{X}_p(t) - \vec{A} \cdot \vec{D}$$

where
$\vec{X}_p$: position of prey
$\vec{A} = 2a \cdot \vec{r}_1 - a$, where $\vec{r}_1 \in [0, 1]$
$\vec{C} = 2 \cdot \vec{r}_2$, where $\vec{r}_2 \in [0, 1]$



Date __/__/__
Page ____

Step 4: Hunting
Wolves update their positions based on α, β, δ:

$$\vec{D}_\alpha = |\vec{C}_1 \cdot \vec{X}_\alpha - \vec{X}|$$
$$\vec{D}_\beta = |\vec{C}_2 \cdot \vec{X}_\beta - \vec{X}|$$
$$\vec{D}_\delta = |\vec{C}_3 \cdot \vec{X}_\delta - \vec{X}|$$
$$\vec{X}_1 = \vec{X}_\alpha - \vec{A}_1 \cdot \vec{D}_\alpha$$
$$\vec{X}_2 = \vec{X}_\beta - \vec{A}_2 \cdot \vec{D}_\beta$$
$$\vec{X}_3 = \vec{X}_\delta - \vec{A}_3 \cdot \vec{D}_\delta$$

~~Step~~ Final position update :

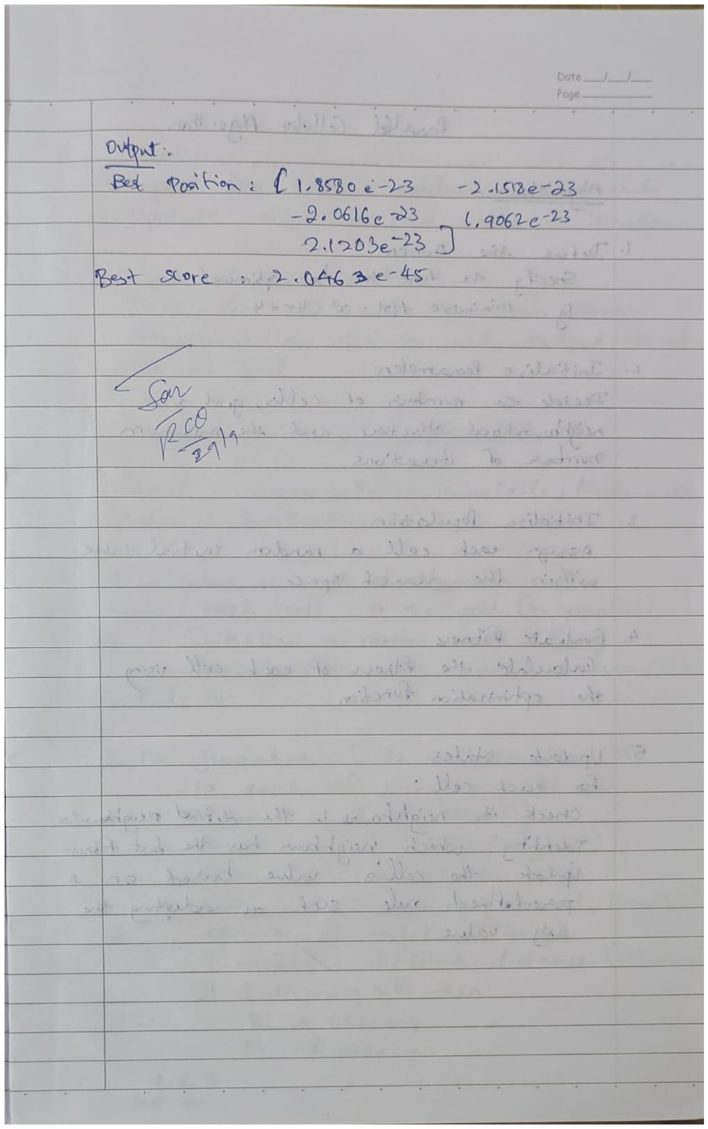
$$\vec{X}(t+1) = \frac{\vec{X}_1 + \vec{X}_2 + \vec{X}_3}{3}$$

Step 5 : Updating the control parameter
do α decreases linearly from 2 to 0 over iteration

$$a = 2 - \frac{2t}{T}$$

Step 6 : Termination
Repeat Steps 2-5 until :
- Max iteration T reached or
- Convergence criterion met.

Return the position of Alpha wolf (α) as best solution.



Date__/__/__
Page____

Output:

```
Best Position : [1.8580e-23      -2.1518e-23
                 -2.0616e-23      1.9062e-23
                  2.1203e-23 ]
Best score    : 2.0463e-45
```

[illegible]

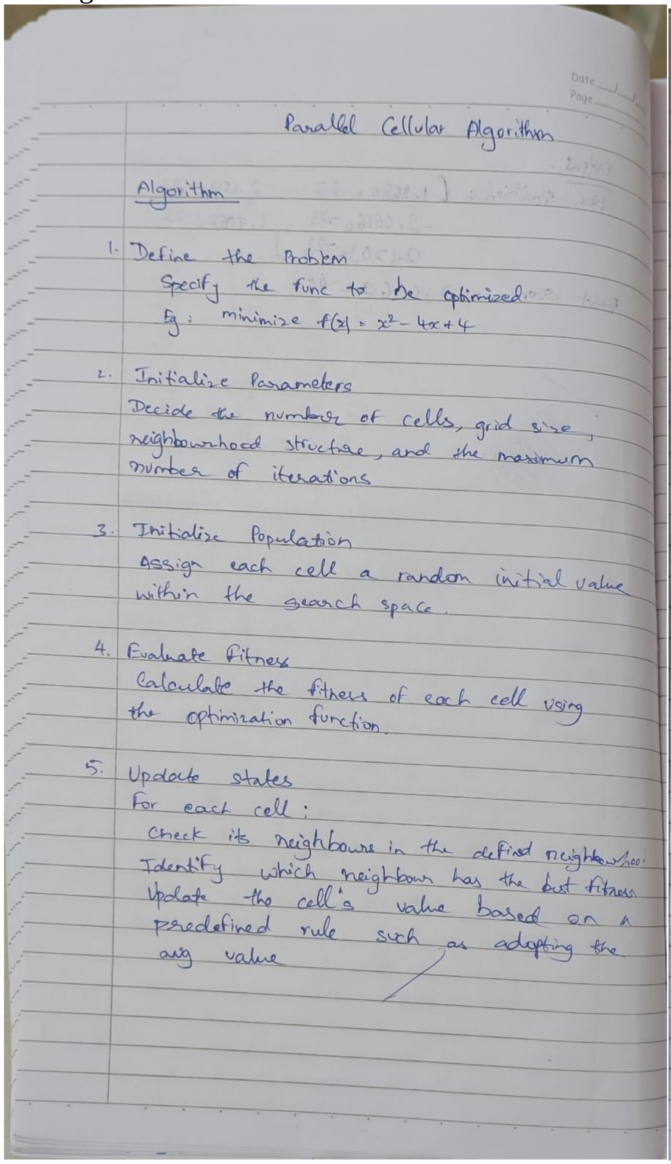
Code:

```
import numpy as np
import random

def distance_matrix(cities):
    n = len(cities)
    dist = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            dist[i][j] = np.linalg.norm(np.array(cities[i]) - np.array(cities[j]))
    return dist

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

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

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

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

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

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

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

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

            new_population.append(new_tour)

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

    return alpha[0], alpha[1]

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

Program 7: Particle Swarm Optimization for Function Optimization:

Particle Swarm Optimization (PSO) is inspired by the social behavior of birds flocking or fish schooling. PSO is used to find optimal solutions by iteratively improving a candidate solution with regard to a given measure of quality. Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:



Date __/__/__
Page ____

Parallel Cellular Algorithm

Algorithm

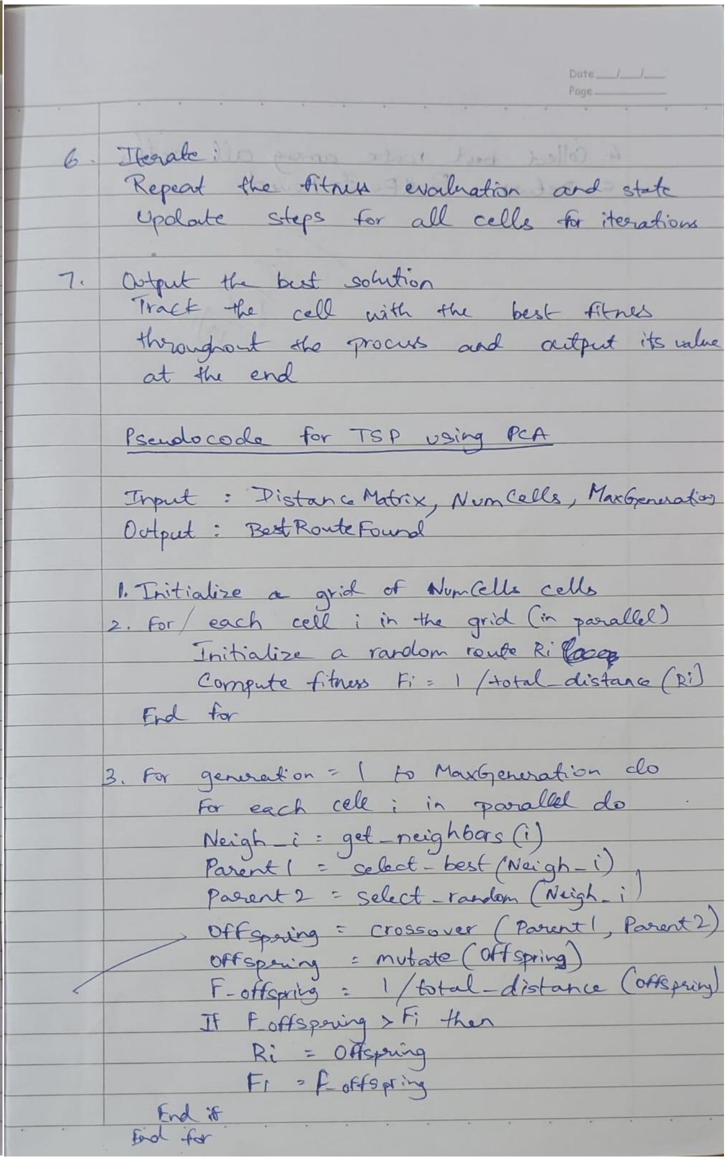
1. Define the Problem
   Specify the func to be optimized.
   Eg : minimize $f(x) = x^2 - 4x + 4$

2. Initialize Parameters
   Decide the number of cells, grid size, neighbourhood structure, and the maximum number of iterations

3. Initialize Population
   Assign each cell a random initial value within the search space.

4. Evaluate Fitness
   Calculate the fitness of each cell using the optimization function.

5. Update states
   For each cell :
   Check its neighbours in the defined neighbourhoo.
   Identify which neighbour has the best fitness
   Update the cell's value based on a predefined rule such as adopting the avg value



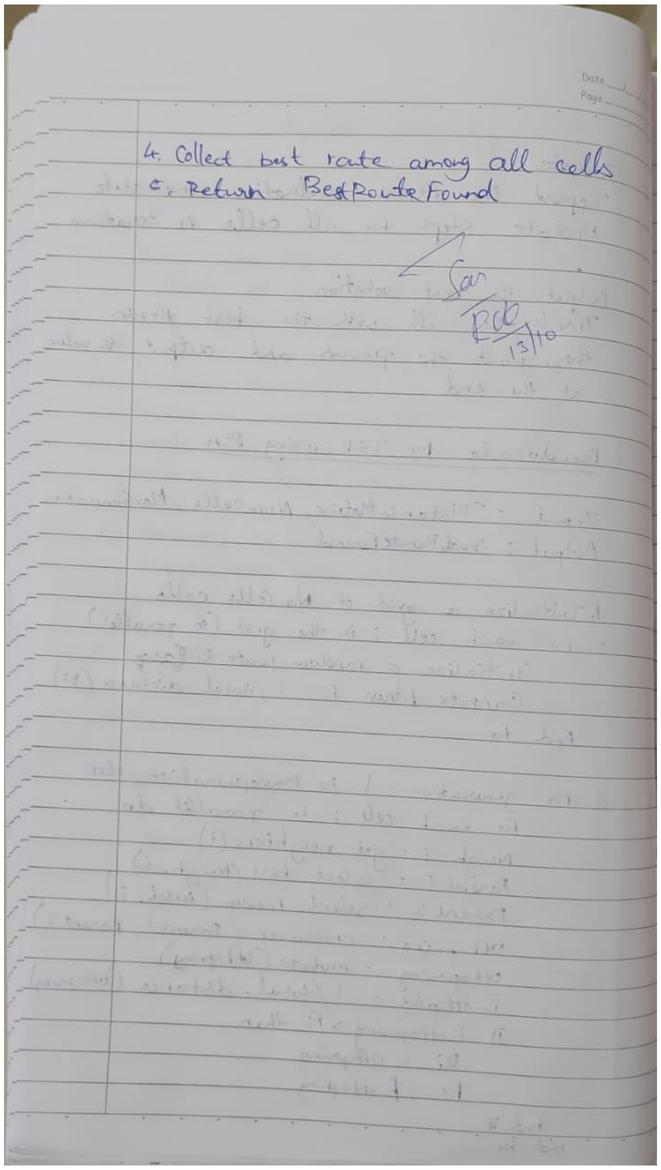
Date __/__/__
Page ____

6. Iterate :
   Repeat the fitness evaluation and state Update steps for all cells for iterations

7. Output the best solution
   Track the cell with the best fitness throughout the process and output its value at the end

Pseudocode for TSP using PCA

```
Input : Distance Matrix, NumCells, MaxGeneration
Output : Best Route Found

1. Initialize a grid of NumCells cells
2. For each cell i in the grid (in parallel)
        Initialize a random route Ri
        Compute fitness Fi = 1 / total_distance (Ri)
   End for

3. For generation = 1 to MaxGeneration do
        For each cell i in parallel do
        Neigh_i = get_neighbors (i)
        Parent 1 = select_best (Neigh_i)
        Parent 2 = select_random (Neigh_i)
        Offspring = crossover (Parent 1, Parent 2)
        Offspring = mutate (Offspring)
        F_offspring = 1 / total_distance (Offspring)
        If F_offspring > Fi then
            Ri = Offspring
            Fi = F_offspring
     End if
   End for
```



4. Collect best rate among all cells
5. Return BestRoute Found

[illegible]
13/10

Code:

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

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

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

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

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

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