VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Sohan A R(1BM22CS285)

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
Sep-2024 to Jan-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

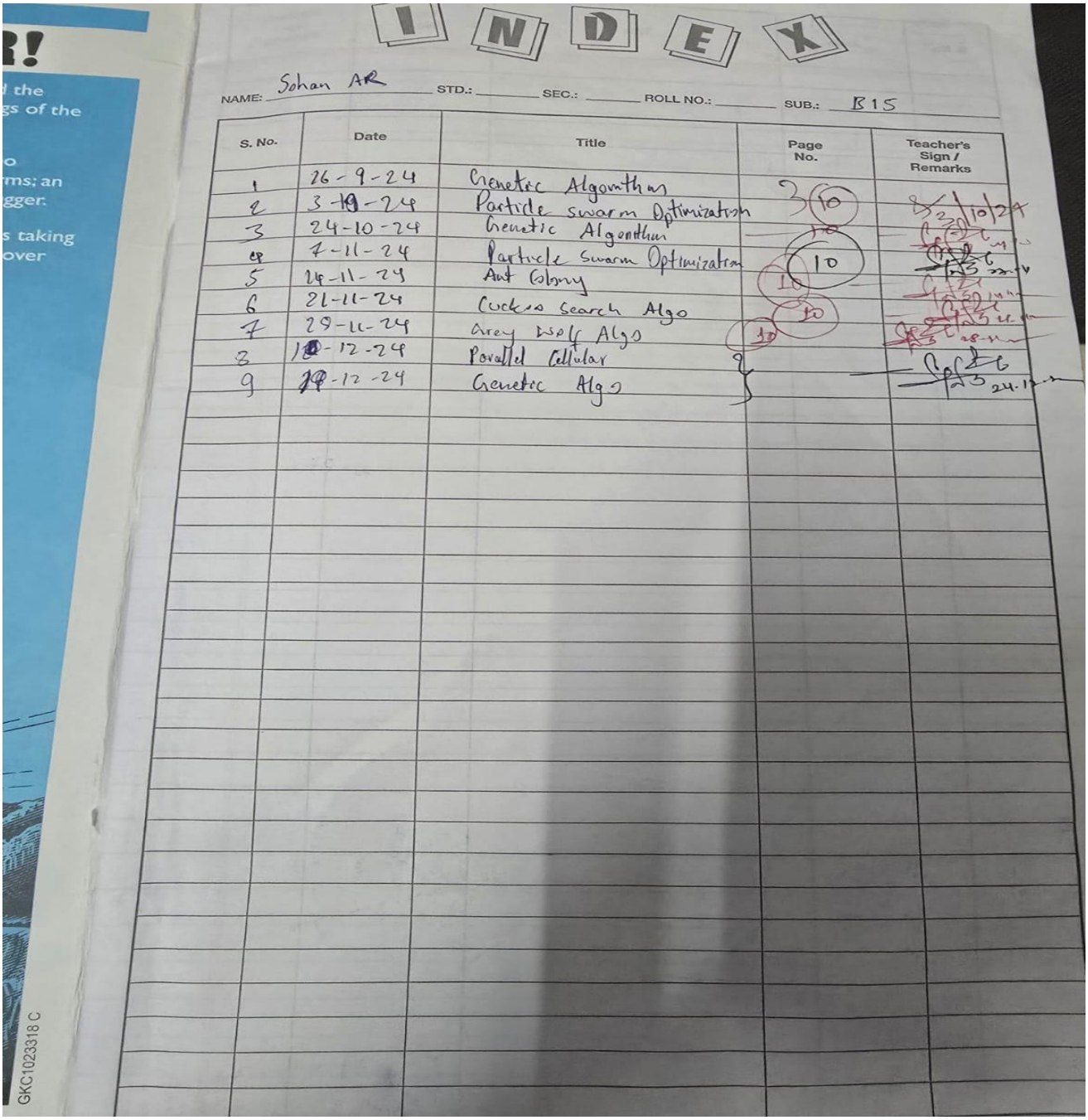
This is to certify that the Lab work entitled " Bio Inspired Systems (23CS5BSBIS)" carried out by **Sohan A R(1BM22CS285),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the 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.

Saritha A N Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link: https://github.com/SohanAR/BIS.git

Index Page of Observation Book:



INDEX

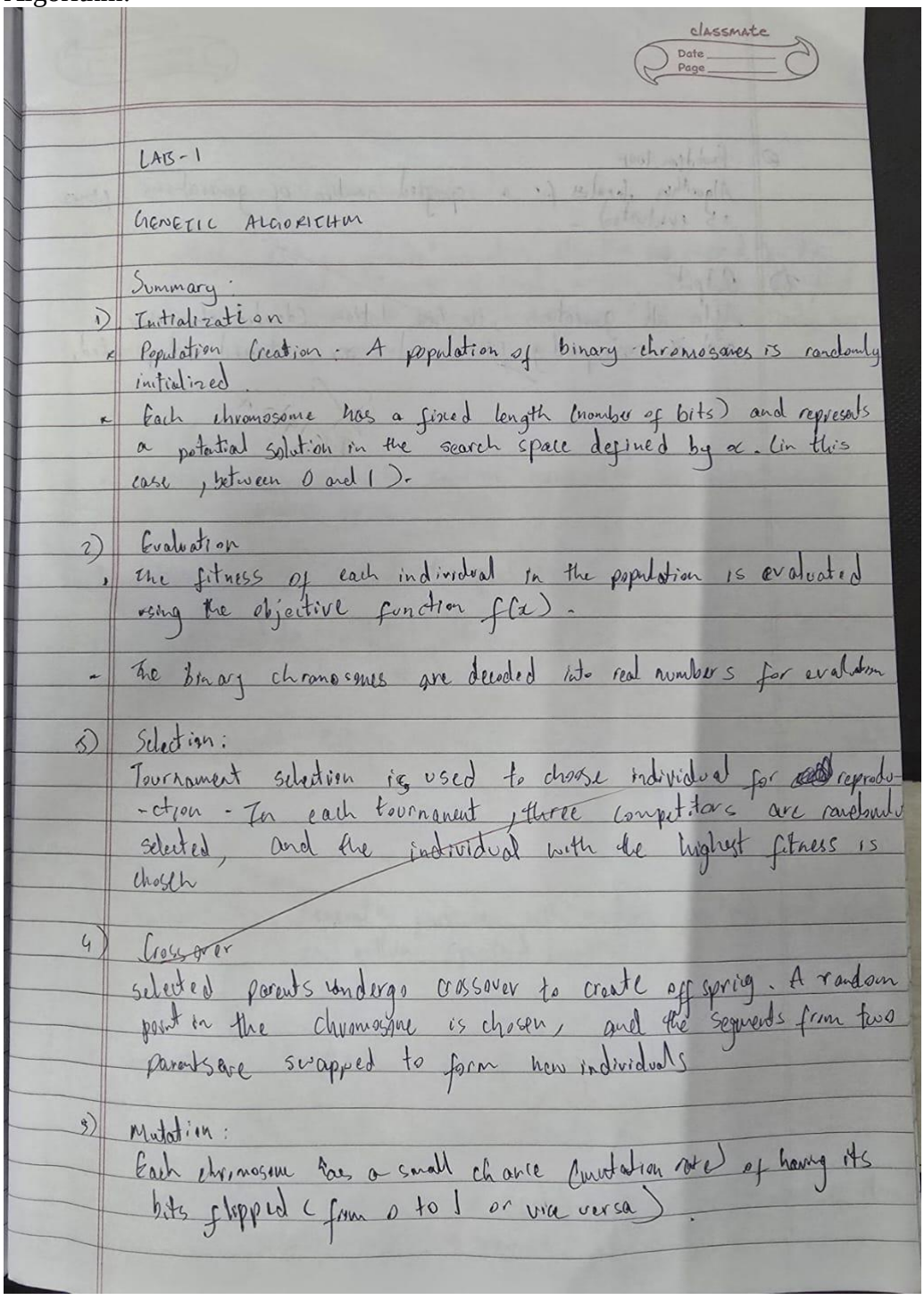
NAME: Sohan AR STD.: _____ SEC.: _____ ROLL NO.: _____ SUB.: B15

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1	26-9-24	Genetic Algorithms	3	
2	3-10-24	Particle swarm Optimization	10	
3	24-10-24	Genetic Algorithm		
4	7-11-24	Particle Swarm Optimization	10	
5	14-11-24	Ant Colony		
6	21-11-24	Cuckoo search Algo		
7	29-11-24	Grey Wolf Algo		
8	10-12-24	Parallel Cellular		
9	17-12-24	Genetic Algo		

Program 1

Problem statement: Optimize the allocation of a portfolio using a Genetic Algorithm to maximize the Sharpe Ratio, balancing expected returns and risk. Ensure the total asset allocation adheres to a fixed budget constraint.

Algorithm:



LAB-1

GENETIC ALGORITHM

Summary:

1) Initialization
- Population Creation: A population of binary chromosomes is randomly initialized.
- Each chromosome has a fixed length (number of bits) and represents a potential solution in the search space defined by α. (in this case, between 0 and 1).

2) Evaluation
- The fitness of each individual in the population is evaluated using the objective function $f(x)$.
- The binary chromosomes are decoded into real numbers for evaluation

3) Selection:
Tournament selection is used to choose individual for reproduction. In each tournament, three competitors are randomly selected, and the individual with the highest fitness is chosen

4) Crossover
selected parents undergo crossover to create offspring. A random point in the chromosome is chosen, and the segments from two parents are swapped to form new individuals

5) Mutation:
Each chromosome has a small chance (mutation rate) of having its bits flipped (from 0 to 1 or vice versa).

6) Evolution loop
Algorithm iterates for a specified number of generations & fitness is evaluated.

7) Output
After all generations, the best solution (decoded chromosome) and its corresponding fitness value are returned and printed.

26/9/24

Code:


```
import random
import numpy as np
import math

def generate_cities(num_cities):
    return [(random.randint(0, 100), random.randint(0, 100)) for _ in range(num_cities)]

def compute_distance_matrix(cities):
    num_cities = len(cities)
    distances = [[0] * num_cities for _ in range(num_cities)]
    for i in range(num_cities):
        for j in range(num_cities):
            if i != j:
                distances[i][j] = math.sqrt(
                    (cities[i][0] - cities[j][0])**2 + (cities[i][1] - cities[j][1])**2
                )
    return distances

class TSP:
    def __init__(self, distances):
        self.distances = distances
        self.num_cities = len(distances)

    def fitness(self, route):

        total_distance = sum(
            self.distances[route[i]][route[i + 1]] for i in range(len(route) - 1)
        )
        total_distance += self.distances[route[-1]][route[0]]

class GeneticAlgorithm:
    def __init__(self, tsp, population_size=100, generations=500, mutation_rate=0.1):
        self.tsp = tsp
        self.population_size = population_size
        self.generations = generations
        self.mutation_rate = mutation_rate
        self.population = self._initialize_population()

    def _initialize_population(self):
        return [random.sample(range(self.tsp.num_cities), self.tsp.num_cities) for _ in range(self.population_

    def _select_parents(self):

        fitnesses = [self.tsp.fitness(route) for route in self.population]
        total_fitness = sum(fitnesses)
        probabilities = [f / total_fitness for f in fitnesses]
        return random.choices(self.population, probabilities, k=2)

    def _crossover(self, parent1, parent2):

        size = len(parent1)
        start, end = sorted(random.sample(range(size), 2))
        child = [-1] * size
        child[start:end] = parent1[start:end]

        p2_idx = 0
        for i in range(size):
            if child[i] == -1:
                while parent2[p2_idx] in child:
                    p2_idx += 1
                child[i] = parent2[p2_idx]
        return child
```

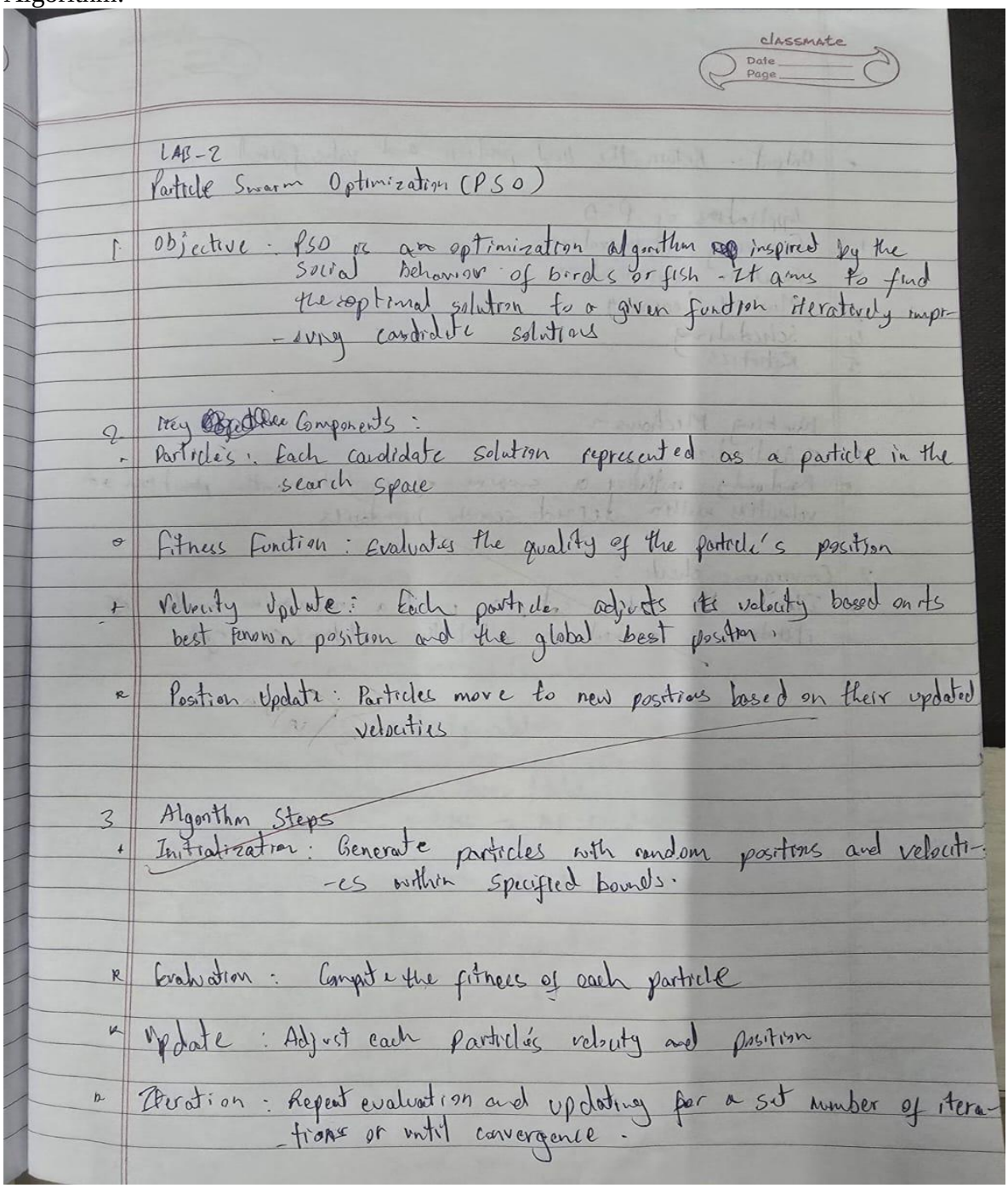
```
    def _mutate(self, route):

        if random.random() < self.mutation_rate:
            i, j = random.sample(range(len(route)), 2)
            route[i], route[j] = route[j], route[i]

    def evolve(self):
        for _ in range(self.generations):
            new_population = []
            for _ in range(self.population_size):
                parent1, parent2 = self._select_parents()
                child = self._crossover(parent1, parent2)
                self._mutate(child)
                new_population.append(child)
            self.population = new_population

    def get_best_solution(self):

        best_route = min(self.population, key=lambda route: 1 / self.tsp.fitness(route))
        best_distance = 1 / self.tsp.fitness(best_route)
        return best_route, best_distance

if __name__ == "__main__":
    num_cities = 5
    cities = generate_cities(num_cities)
    distances = compute_distance_matrix(cities)

    print("City Coordinates:")
    for i, city in enumerate(cities):
        print(f"City {i}: {city}")

    tsp = TSP(distances)
    ga = GeneticAlgorithm(tsp, population_size=50, generations=100, mutation_rate=0.2)
    ga.evolve()
    best_route, best_distance = ga.get_best_solution()

    print("\nBest route:", best_route)
    print("Best distance:", best_distance)
```

```
City Coordinates:
City 0: (22, 31)
City 1: (33, 46)
City 2: (89, 0)
City 3: (65, 0)
City 4: (3, 17)

Best route: [2, 1, 0, 4, 3]
Best distance: 202.96101904990562
```

Program 2

Problem statement: Implement a Particle Swarm Optimization (PSO) algorithm to minimize benchmark functions, such as the Rastrigin and Sphere functions, by optimizing their input parameters. The goal is to find the global minimum while efficiently exploring the solution space using swarm intelligence.

Algorithm:



LAB-2
Particle Swarm Optimization (PSO)

1. Objective - PSO is an optimization algorithm inspired by the social behaviour of birds or fish - It aims to find the optimal solution to a given function iteratively improving candidate solutions

2. Key Components :
- Particles : Each candidate solution represented as a particle in the search space
- Fitness Function : Evaluates the quality of the particle's position
- Velocity Update : Each particle adjusts its velocity based on its best known position and the global best position.
- Position Update : Particles move to new positions based on their updated velocities

3. Algorithm Steps
- Initialization : Generate particles with random positions and velocities within specified bounds.
- Evaluation : Compute the fitness of each particle
- Update : Adjust each particle's velocity and position
- Iteration : Repeat evaluation and updating for a set number of iterations or until convergence.

- Output :- Return the best position and value found

Applications of PSO

1. Engineering Optimization
2. Function Optimization
3. Machine Learning
4. Scheduling
5. Robotics

Working Mechanism

1. Initialization
   - Randomly initialize a swarm of particles with position and velocities within defined search boundaries

2. Convergence check :
   - Iterate the above steps until a maximum number of iterations is reached or until the solution converges.

Code:


```
import random
import numpy as np

def objective_function(position):
    """The function to be minimized."""
    x, y = position
    return x**2 + y**2

def pso(objective_function, dimensions, iterations, population_size, w=0.7, c1=1.4, c2=1.4):
    """
    Particle Swarm Optimization algorithm.

    Args:
        objective_function: The function to be minimized.
        dimensions: The number of dimensions of the search space.
        iterations: The number of iterations to run the algorithm.
        population_size: The number of particles in the swarm.
        w: Inertia weight.
        c1: Cognitive parameter.
        c2: Social parameter.

    Returns:
        A tuple containing the best solution found and its corresponding objective function value.
    """
    particles = []
    for _ in range(population_size):
        position = np.random.uniform(-10, 10, dimensions)
        velocity = np.random.uniform(-1, 1, dimensions)
        particles.append({
            'position': position,
            'velocity': velocity,
            'best_position': position.copy(),
            'best_value': objective_function(position)
        })

    global_best_position = particles[0]['best_position'].copy()
    global_best_value = particles[0]['best_value']

    for _ in range(iterations):
        for particle in particles:

            r1 = random.random()
            r2 = random.random()
            particle['velocity'] = (w * particle['velocity'] +
                                    c1 * r1 * (particle['best_position'] - particle['position']) +
                                    c2 * r2 * (global_best_position - particle['position']))

            particle['position'] = particle['position'] + particle['velocity']

            particle['position'] = np.clip(particle['position'], -10, 10)

            value = objective_function(particle['position'])

            if value < particle['best_value']:
                particle['best_value'] = value
                particle['best_position'] = particle['position'].copy()

            if value < global_best_value:
                global_best_value = value
                global_best_position = particle['position'].copy()

    return global_best_position, global_best_value
```



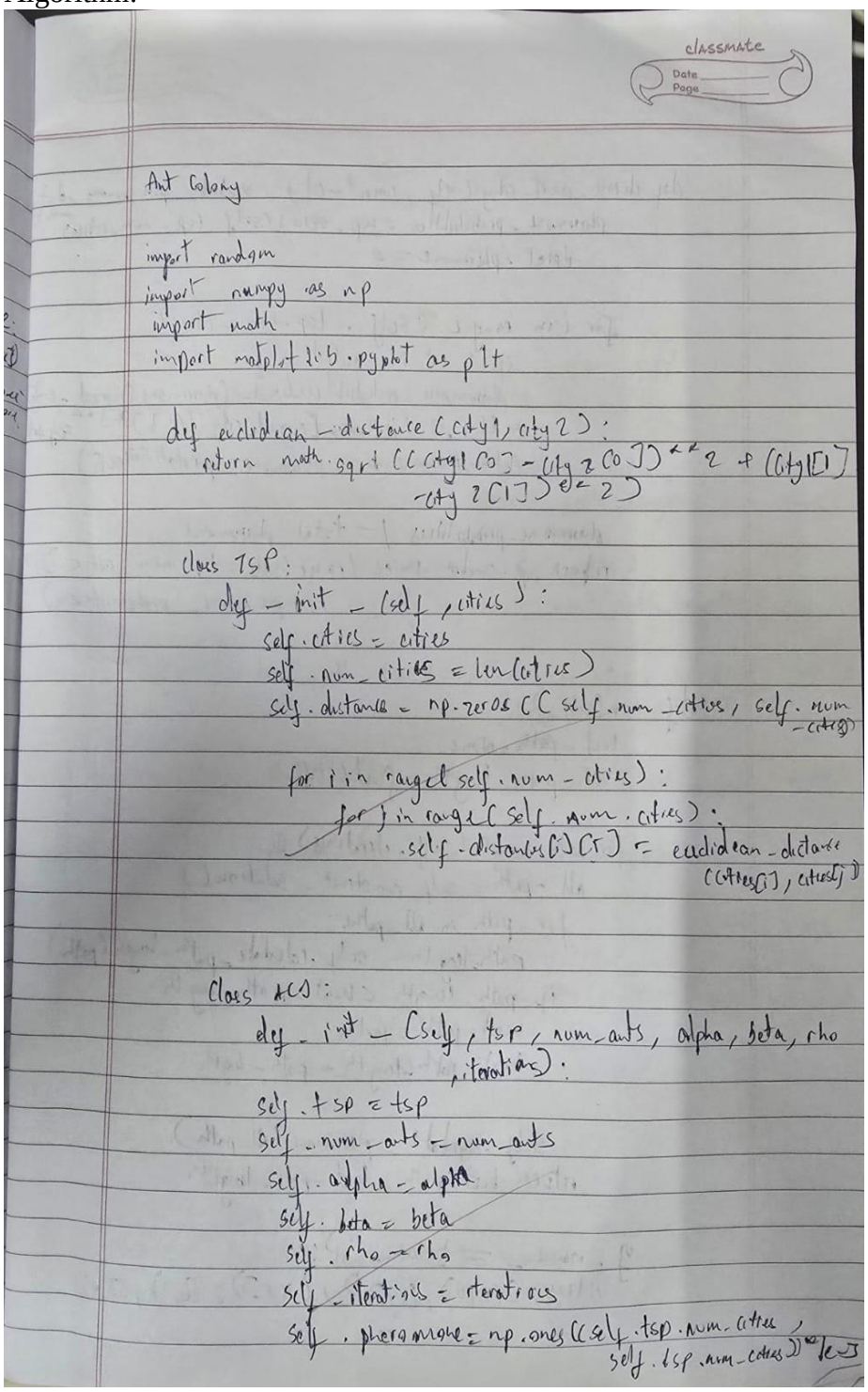
```
dimensions = 2
iterations = 100
population_size = 50

best_position, best_value = pso(objective_function, dimensions, iterations, population_size)
print(f"Best position found: {best_position}")
print(f"Best value found: {best_value}")
```

```
Best position found: [ 4.04789703e-08 -2.23363404e-08]
Best value found: 2.137459138638845e-15
```

Program 3

Problem statement: Implement an Ant Colony Optimization (ACO) algorithm to solve the Traveling Salesman Problem (TSP), where the goal is to find the shortest possible path that visits all cities exactly once and returns to the starting city. The algorithm should utilize pheromone trails and heuristic information to guide the search efficiently.

Algorithm:



```
Ant Colony

import random
import numpy as np
import math
import matplotlib.pyplot as plt

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

class TSP:
    def __init__(self, cities):
        self.cities = cities
        self.num_cities = len(cities)
        self.distances = np.zeros((self.num_cities, self.num
                                   _cities))

        for i in range(self.num_cities):
            for j in range(self.num_cities):
                self.distances[i][j] = euclidean_distance
                                       (cities[i], cities[j])


class ACO:
    def __init__(self, tsp, num_ants, alpha, beta, rho
                 , iterations):
        self.tsp = tsp
        self.num_ants = num_ants
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.iterations = iterations
        self.pheromone = np.ones((self.tsp.num_cities,
                                  self.tsp.num_cities))/len
```

```
def choose_next_city(self, current_city, visited, pheromone):
    pheromone_probabilities = np.zeros(self.tsp.num_cities)
    total_pheromone = 0

    for i in range(self.tsp.num_cities):
        if i not in visited:
            pheromone_probabilities[i] = (pheromone[current_city][i] ** self.alpha) * ((1.0 / distances[current_city][i]) ** self.beta)
            total_pheromone += pheromone_probabilities[i]

    pheromone_probabilities /= total_pheromone
    return np.random.choice(range(self.tsp.num_cities), p=pheromone_probabilities)

def run(self):
    best_path = None
    best_path_length = float('inf')

    for _ in range(self.iterations):
        all_paths = self.construct_solutions()
        for path in all_paths:
            path_length = self.calculate_path_length(path)
            if path_length < best_path_length:
                best_path = path
                best_path_length = path_length

        self.update_pheromone(all_paths)
    return best_path, best_path_length

if __name__ == "__main__":
    cities = [(0,0), (1,3), (4,3), (6,1), (3,0)]
```

```
tsp = TSP(cities)

num_ants = 10
alpha = 1.0
beta = 2.0
rho = 0.1
Iterations = 100

aco = ACO(tsp, num_ants, alpha, beta, rho, Iterations)
best_path, best_path_length = aco.run()

print("Best Path:", best_path)
print("Best path length:", best_path_length)

best_path_coords = [cities[i] for i in best_path]
                   +[cities[best_path[0]]]

best_path_coords = np.array(best_path_coords)

plt.plot(best_path_coords[:,0], best_path_coords
                                [:,1],mo
                                =

plt.scatter(*zip(*cities), color='red')
plt.title(f"Best TSP Path (length: {best_path_leng
                                  :2f})"

plt.show()
```

Output:

Best path: [1, 2, 3, 4, 0]
Best Path length: 15.157987274

14.11

Code:


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

# Define constants for the algorithm
NUM_ANTS = 50
NUM_CITIES = 20  # Now we have 20 cities
ALPHA = 1.0  # Influence of pheromone
BETA = 2.0   # Influence of distance
RHO = 0.1    # Pheromone evaporation rate
Q = 100       # Pheromone deposit constant
MAX_ITER = 100  # Maximum number of iterations

# Predefined 20 cities (coordinates in 2D space)
def generate_cities():
    cities = np.array([
        [5, 10], [11, 5], [14, 9], [12, 15], [8, 13],  # Cities 0-4
        [10, 10], [13, 7], [16, 5], [14, 3], [18, 6],  # Cities 5-9
        [4, 2], [7, 1], [8, 5], [6, 7], [4, 10],  # Cities 10-14
        [15, 18], [12, 17], [3, 18], [17, 12], [19, 8]  # Cities 15-19
    ])
    return cities

# Compute the distance matrix
def compute_distance_matrix(cities):
    num_cities = len(cities)
    distance_matrix = np.zeros((num_cities, num_cities))
    for i in range(num_cities):
        for j in range(i + 1, num_cities):
            dist = np.linalg.norm(cities[i] - cities[j])
            distance_matrix[i, j] = dist
            distance_matrix[j, i] = dist
    return distance_matrix

# Initialize pheromone matrix
def initialize_pheromone_matrix(num_cities):
    pheromone_matrix = np.ones((num_cities, num_cities))  # Pheromone starts as 1 for all edges
    np.fill_diagonal(pheromone_matrix, 0)  # No pheromone on the diagonal (self-loops)
    return pheromone_matrix

# Calculate the total length of a tour
def calculate_tour_length(tour, dist_matrix):
    length = 0
    for i in range(len(tour) - 1):
        length += dist_matrix[tour[i], tour[i + 1]]
    length += dist_matrix[tour[-1], tour[0]]  # Returning to the start
    return length

# Ant solution construction (probabilistic decision on next city)
def construct_solution(num_cities, pheromone_matrix, dist_matrix):
    tour = [random.randint(0, num_cities - 1)]  # Start from a random city
    visited = set(tour)

    while len(tour) < num_cities:
        current_city = tour[-1]
        probabilities = []
        for next_city in range(num_cities):
            if next_city not in visited:
                pheromone = pheromone_matrix[current_city, next_city] ** ALPHA
                distance = (1.0 / dist_matrix[current_city, next_city]) ** BETA
                probabilities.append(pheromone * distance)
            else:
                probabilities.append(0)
```

```
            total_prob = sum(probabilities)
            probabilities = [p / total_prob for p in probabilities]

            # Choose the next city based on the probabilities
            next_city = np.random.choice(range(num_cities), p=probabilities)
            tour.append(next_city)
            visited.add(next_city)

        return tour

    # Update the pheromone matrix based on the solutions found by ants
    def update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour):
        # Evaporate pheromone
        pheromone_matrix *= (1 - RHO)

        # Add pheromone for all ants
        for tour in all_tours:
            tour_length = calculate_tour_length(tour, dist_matrix)
            for i in range(len(tour) - 1):
                pheromone_matrix[tour[i], tour[i + 1]] += Q / tour_length
            pheromone_matrix[tour[-1], tour[0]] += Q / calculate_tour_length(tour, dist_matrix)

        # Add pheromone for the best tour
        best_length = calculate_tour_length(best_tour, dist_matrix)
        for i in range(len(best_tour) - 1):
            pheromone_matrix[best_tour[i], best_tour[i + 1]] += Q / best_length
        pheromone_matrix[best_tour[-1], best_tour[0]] += Q / best_length

    # Main ACO algorithm for solving TSP
    def ant_colony_optimization(num_cities, dist_matrix, pheromone_matrix, max_iter):
        best_tour = None
        best_tour_length = float('inf')

        # Main loop
        for iteration in range(max_iter):
            all_tours = []

            # Step 1: All ants construct their solutions
            for _ in range(NUM_ANTS):
                tour = construct_solution(num_cities, pheromone_matrix, dist_matrix)
                all_tours.append(tour)
                tour_length = calculate_tour_length(tour, dist_matrix)

                # Step 2: Update the best tour if necessary
                if tour_length < best_tour_length:
                    best_tour = tour
                    best_tour_length = tour_length

            # Step 3: Update pheromone matrix
            update_pheromone(pheromone_matrix, all_tours, dist_matrix, best_tour)

            # Optional: print progress every 10 iterations
            if iteration % 10 == 0:
                print(f"Iteration {iteration}: Best tour length = {best_tour_length:.2f}")

        return best_tour, best_tour_length


    # Main Execution
    if __name__ == "__main__":
        # Step 1: Generate predefined cities and distance matrix
        cities = generate_cities()
        dist_matrix = compute_distance_matrix(cities)
```

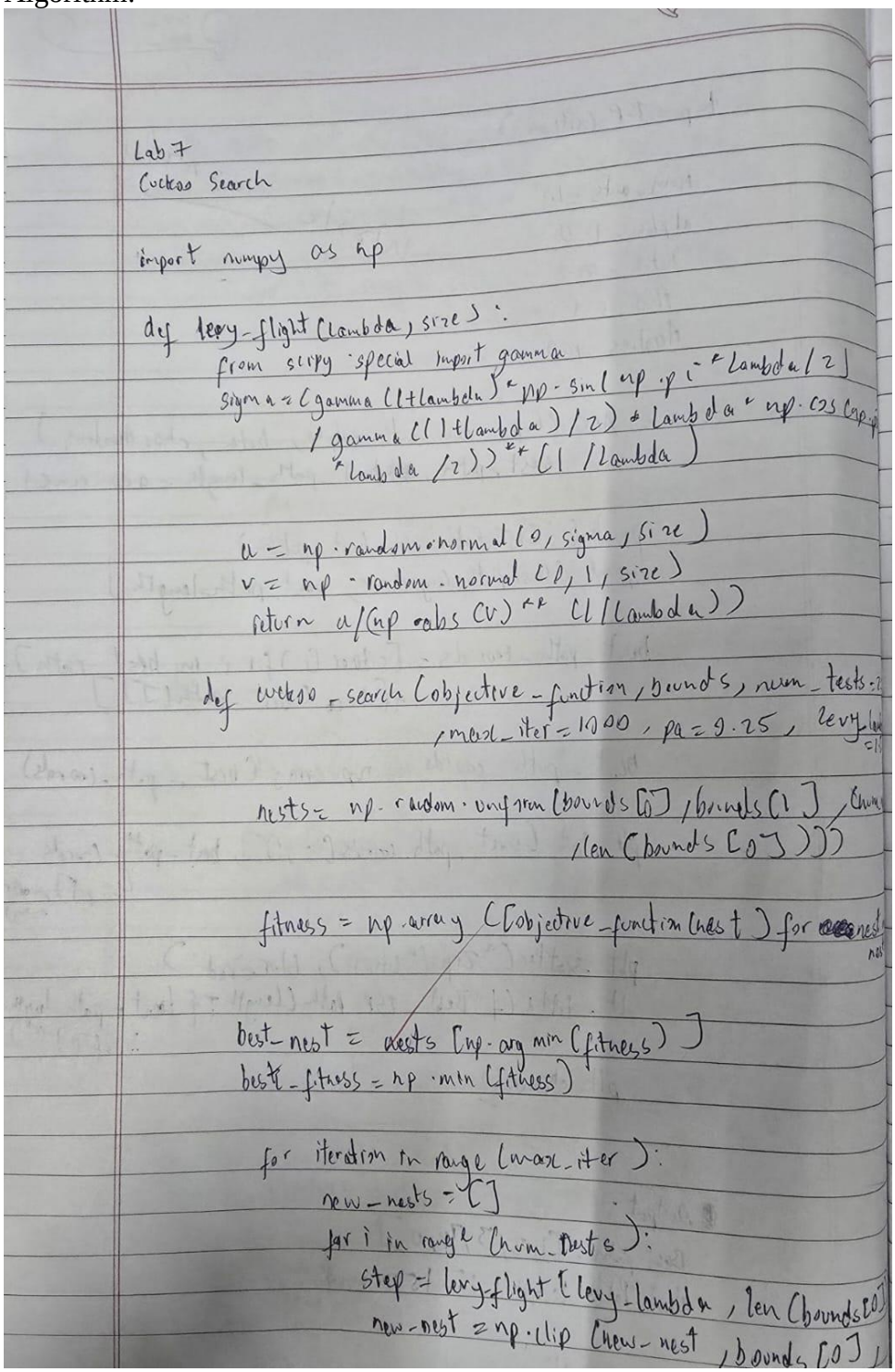
```
# Step 2: Initialize pheromone matrix
pheromone_matrix = initialize_pheromone_matrix(NUM_CITIES)

# Step 3: Run ACO algorithm
best_tour, best_tour_length = ant_colony_optimization(NUM_CITIES, dist_matrix, pheromone_matrix, MAX_ITER)

# Step 4: Output the best tour and visualize it
print(f"Best tour length: {best_tour_length:.2f}")
```

```
Iteration 0: Best tour length = 107.48
Iteration 10: Best tour length = 81.48
Iteration 20: Best tour length = 80.59
Iteration 30: Best tour length = 80.50
Iteration 40: Best tour length = 79.23
Iteration 50: Best tour length = 79.23
Iteration 60: Best tour length = 77.88
Iteration 70: Best tour length = 77.88
Iteration 80: Best tour length = 77.88
Iteration 90: Best tour length = 77.88
Best tour length: 77.88
```

Program 4

Problem statement: Implement the Cuckoo Search Algorithm for feature selection to identify an optimal subset of features that maximizes the classification accuracy of a Support Vector Machine (SVM) on a given dataset.

Algorithm:



```
Lab 7
Cuckoo Search

import numpy as np

def levy_flight(lambda, size):
    from scipy.special import gamma
    sigma = (gamma(1+lambda) * np.sin(np.pi * lambda/2)
        / gamma((1+lambda)/2) * lambda * np.cos(np.pi
        * lambda/2)) ** (1/lambda)

    u = np.random.normal(0, sigma, size)
    v = np.random.normal(0, 1, size)
    return u/(np.abs(v) ** (1/lambda))

def cuckoo_search(objective_function, bounds, num_tests=
        , max_iter=1000, pa=0.25, levy_lam
        =1)

    nests = np.random.uniform(bounds[0], bounds[1], (num
        , len(bounds[0])))

    fitness = np.array([objective_function(nest) for nest
        in nests])

    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        new_nests = []
        for i in range(num_nests):
            step = levy_flight(levy_lambda, len(bounds[0])
            new_nest = np.clip(new_nest, bounds[0],
```

```
new_nests.append(new_nest)

new_fitness = np.array([objective_function(nest) for nest in
                                        new_nests])
for i in range(num_nests):
    if new_fitness[i] < fitness[i]:
        nests[i] = new_nests[i]
        fitness[i] = new_fitness[i]


for i in range(num_nests):
    if np.random.rand() < pa:
        nests[i] = np.random.uniform(bounds[0], bounds[1]
                                    , len(bounds[0]))
        fitness[i] = objective_function(nests[i])


current_best_fitness = np.min(fitness)
if current_best_fitness < best_fitness:
    best_fitness = current_best_fitness
    best_nest = nests[np.argmin(fitness)]

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

return best_nest, best_fitness

bounds = np.array([[-5,-5], [5,5]])

best_solution, best_objective_value = cuckoo_search(objective
                                        _function, bounds)
print("Best Solution", best_solution)
print("Best Objective Value:", best_objective_value)
```

Output:

Best Solution: [-0.51597054 -1.41565335]

Best Objective Value: 0.002085051461112157 2

Code:


```
#cuckoo search(Traffic Signal Optimization)
import numpy as np
from scipy.special import gamma

def fitness_function(x):
    waiting_times = np.array([10 + (x[i] ** 2) / 100 for i in range(len(x))])
    total_waiting_time = np.sum(waiting_times)
    return total_waiting_time

def levy_flight(dim, beta=1.5):
    sigma_u = np.power((gamma(1 + beta) * np.sin(np.pi * beta / 2) /
                        gamma((1 + beta) / 2) * beta * (2 ** (beta - 1))), 1 / beta)
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, 1, dim)
    step = u / np.power(np.abs(v), 1 / beta)
    return step

def cuckoo_search(dim, bounds, num_nests, max_iter, p_a=0.25, Lambda=1.5):
    nests = np.random.uniform(bounds[0], bounds[1], (num_nests, dim))
    fitness = np.array([fitness_function(nest) for nest in nests])

    best_idx = np.argmin(fitness)
    best_nest = nests[best_idx]
    best_fitness = fitness[best_idx]

    for iter in range(max_iter):
        new_nests = np.copy(nests)
        for i in range(num_nests):
            step = levy_flight(dim, Lambda)
            new_nests[i] = nests[i] + step
            new_nests[i] = np.clip(new_nests[i], bounds[0], bounds[1])

        new_fitness = np.array([fitness_function(nest) for nest in new_nests])

        for i in range(num_nests):
            if new_fitness[i] < fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        if np.random.rand() < p_a:
            random_idx = np.random.randint(num_nests)
            nests[random_idx] = np.random.uniform(bounds[0], bounds[1], dim)
            fitness[random_idx] = fitness_function(nests[random_idx])

        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]

        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]



    return best_nest, best_fitness

dim = 3
bounds = [10, 120]
num_nests = 20
max_iter = 100

best_solution, best_value = cuckoo_search(dim, bounds, num_nests, max_iter)

print("\n--- Best Solution ---")
```



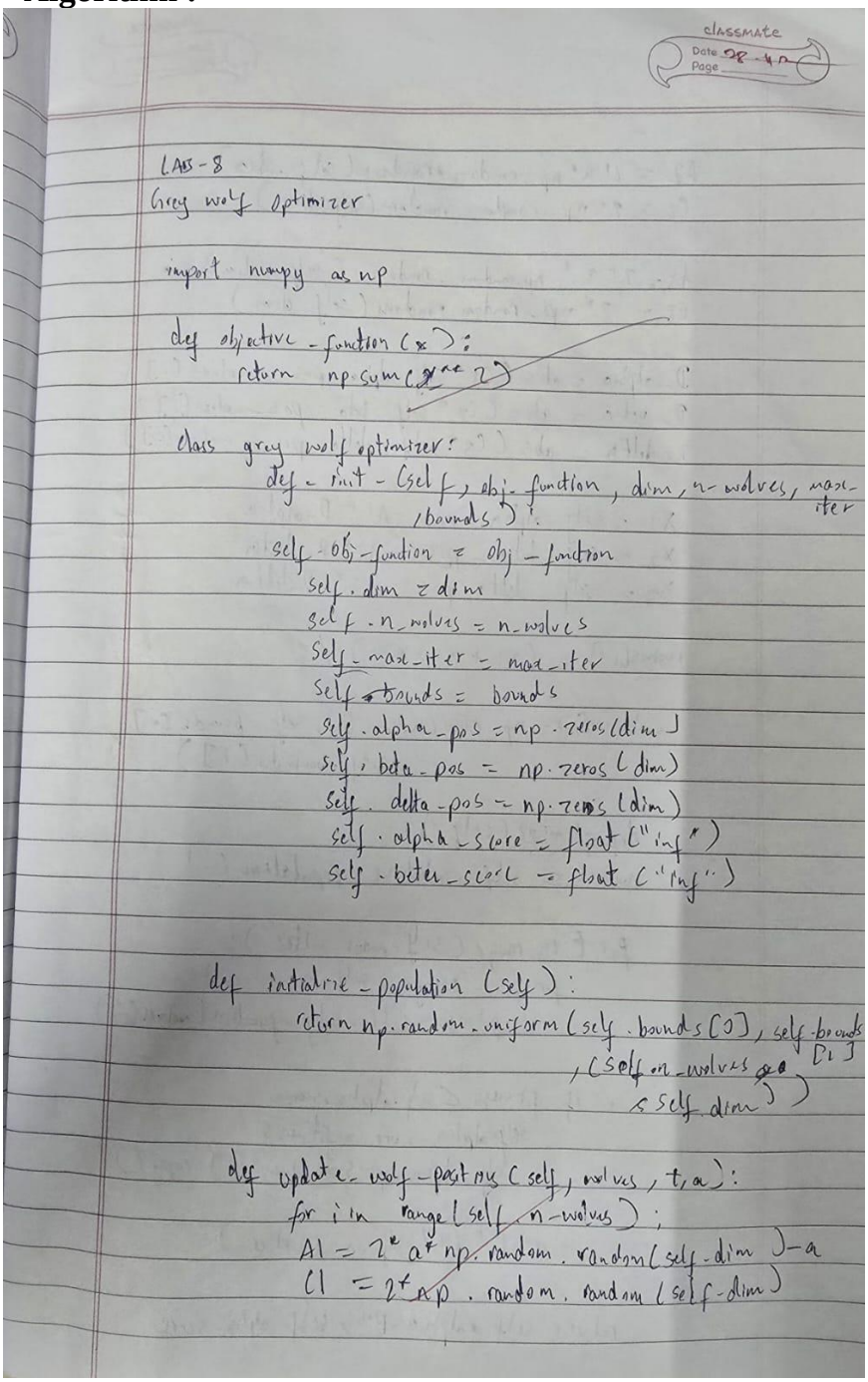
```
print("Green Light Timings (seconds):", best_solution)
print("Best Fitness Value (Total Waiting Time):", best_value)
```

```
--- Best Solution ---
Green Light Timings (seconds): [10. 10. 10.]
Best Fitness Value (Total Waiting Time): 33.0
```

Program 5

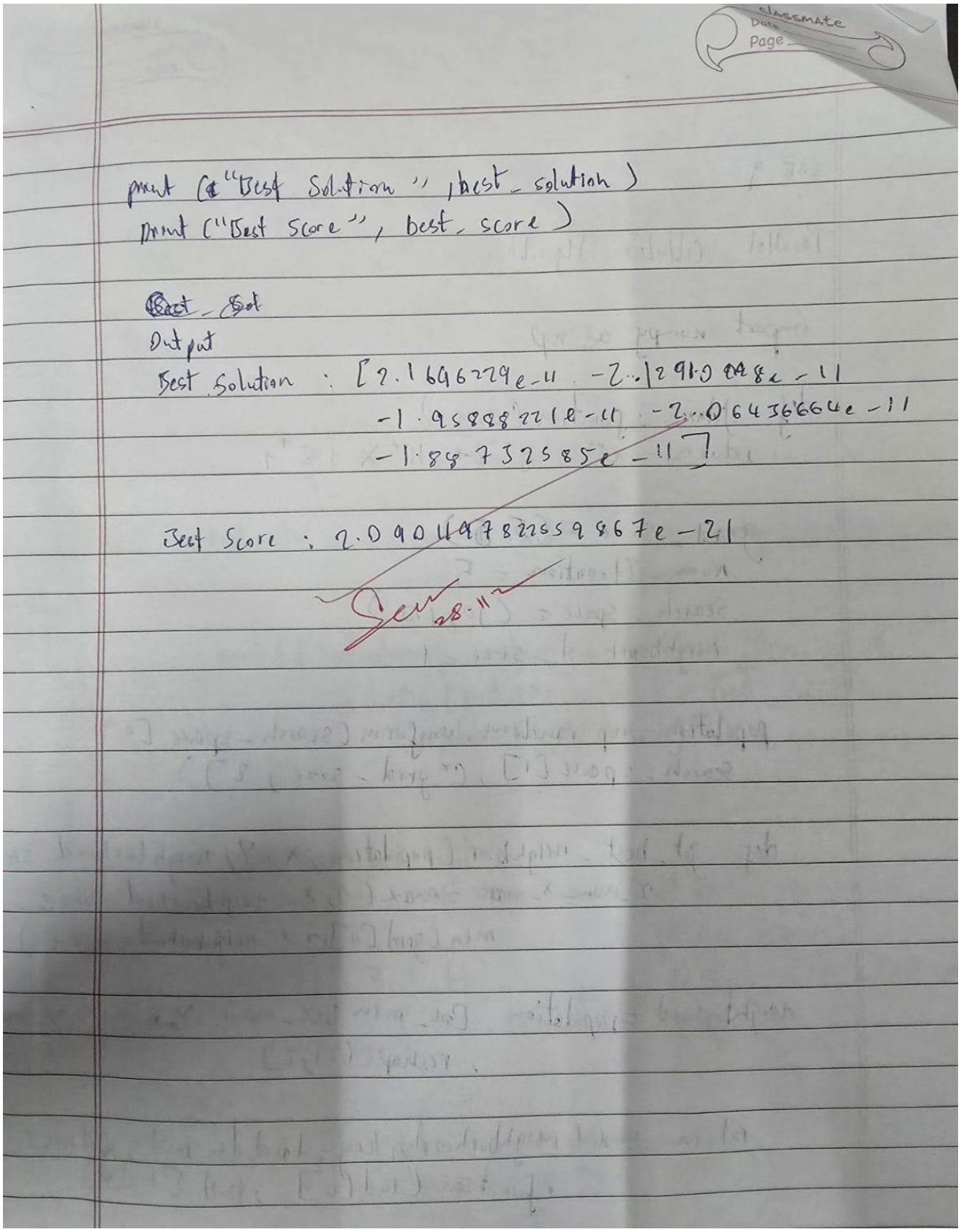
Problem Statement : Implement the Grey Wolf Optimizer (GWO) to optimize the hyperparameters (C and gamma) of a Support Vector Machine (SVM) classifier for achieving the best classification accuracy on the Iris dataset.

Algorithm :



```
LAB-8
Grey wolf optimizer

import numpy as np

def objective_function(x):
    return np.sum(x**2)

class grey_wolf_optimizer:
    def __init__(self, obj_function, dim, n_wolves, max_iter
                 , bounds):
        self.obj_function = obj_function
        self.dim = dim
        self.n_wolves = n_wolves
        self.max_iter = max_iter
        self.bounds = bounds
        self.alpha_pos = np.zeros(dim)
        self.beta_pos = np.zeros(dim)
        self.delta_pos = np.zeros(dim)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")

def initialize_population(self):
    return np.random.uniform(self.bounds[0], self.bounds
                             [1]
                             , (self.n_wolves
                             , self.dim))

def update_wolf_positions(self, wolves, t, a):
    for i in range(self.n_wolves):
        A1 = 2*a*np.random.random(self.dim)-a
        C1 = 2*np.random.random(self.dim)
```

```
A2 = 2*a*np.random.random(self.dim) - a
C2 = 2*np.random.random(self.dim)

A3 = 2*a*np.random.random(self.dim) - a
C3 = 2*np.random.random(self.dim)

D_alpha = abs(C1*self.alpha_pos - wolves[i])
D_beta = abs(C2*self.beta_pos - wolves[i])
D_delta = abs(C3*self.delta_pos - wolves[i])

X1 = self.alpha_pos - A1*D_alpha
X2 = self.beta_pos - A2*D_beta
X3 = self.delta_pos - A3*D_delta

wolves[i] = (X1 + X2 + X3)/3

wolves[i] = np.clip(wolves[i], self.bounds[0]
                    , self.bounds[1])

def optimize(self):
wolves = self.initialize_population()

for t in range(self.max_iter):
    for i in range(self.n_wolves):
        fitness = self.obj_function(wolves[i])

        if fitness < self.alpha_scores:
            self.alpha_score = fitness
            self.alpha_pos = wolves[i].copy()

    a = 2 - t*(2/self.max_iter)

return self.alpha_pos, self.alpha_score
```


print ("Best Solution ", best_solution)
print ("Best score", best_score)
Best_Sol
Output
Best Solution : [2.1696229e-11 -2.1291048e-11
-1.9588827218-11 -2.0643666 4e-11
-1.88 7325 85e-11]
Best Score : 2.0901147822559867e-21
Sen 28.11

Code:

```
import numpy as np


N_INPUTS = 3
N_HIDDEN = 5
N_OUTPUTS = 1
N_WOLVES = 30
MAX_ITER = 100
LB = -10.0
UB = 10.0

def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)


def forward_pass(input_data, weights):
    hidden_layer = np.dot(input_data, weights[:N_INPUTS *
N_HIDDEN].reshape(N_INPUTS, N_HIDDEN))
    hidden_layer = sigmoid(hidden_layer)
    output = np.dot(hidden_layer, weights[N_INPUTS *
N_HIDDEN:].reshape(N_HIDDEN, N_OUTPUTS))
    return output, hidden_layer

def fitness_function(weights, inputs, targets, n_samples):
    total_error = 0.0
    for i in range(n_samples):
        output, _ = forward_pass(inputs[i], weights)
        total_error += (output - targets[i]) ** 2
    return total_error / n_samples


def rand_range(min_val, max_val):
    return min_val + (max_val - min_val) * np.random.random()

def update_position(positions, alpha_pos, beta_pos, delta_pos, i, t):
    A = 2 - t * (2.0 / MAX_ITER)
    C = 2 * np.random.random()

    for j in range(len(positions[i])):

        D_alpha = np.abs(C * alpha_pos[j] - positions[i][j])
        D_beta = np.abs(C * beta_pos[j] - positions[i][j])
        D_delta = np.abs(C * delta_pos[j] - positions[i][j])

        new_position = alpha_pos[j] - A * D_alpha if np.random.random()
> 0.5 else beta_pos[j] - A * D_beta
```

```
        positions[i][j] = np.clip(new_position, LB, UB)


def gwo_optimization(inputs, targets, n_samples):
    positions = np.random.uniform(LB, UB, (N_WOLVES, N_INPUTS *
N_HIDDEN + N_HIDDEN))  # Wolves' positions (weights)
    fitness = np.zeros(N_WOLVES)  # Fitness of wolves
    alpha_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)  # Best
position (alpha wolf)
    beta_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)   # Second best
position (beta wolf)
    delta_pos = np.zeros(N_INPUTS * N_HIDDEN + N_HIDDEN)  # Third best
position (delta wolf)
    alpha_score = float('inf')  # Best fitness value (alpha wolf)
    beta_score = float('inf')   # Second best fitness value (beta wolf)
    delta_score = float('inf')  # Third best fitness value (delta wolf)


    for i in range(N_WOLVES):
        fitness[i] = fitness_function(positions[i], inputs, targets,
n_samples)


        if fitness[i] < alpha_score:
            alpha_score = fitness[i]
            alpha_pos = positions[i]
        elif fitness[i] < beta_score:
            beta_score = fitness[i]
            beta_pos = positions[i]
        elif fitness[i] < delta_score:
            delta_score = fitness[i]
            delta_pos = positions[i]

    for t in range(MAX_ITER):
        for i in range(N_WOLVES):
            update_position(positions, alpha_pos, beta_pos, delta_pos,
i, t)
            fitness[i] = fitness_function(positions[i], inputs,
targets, n_samples)


            if fitness[i] < alpha_score:
                alpha_score = fitness[i]
                alpha_pos = positions[i]
            elif fitness[i] < beta_score:
                beta_score = fitness[i]
                beta_pos = positions[i]
```

```
            elif fitness[i] < delta_score:
                delta_score = fitness[i]
                delta_pos = positions[i]



        if t % 10 == 0:
            print(f"Iteration {t}/{MAX_ITER}, Best Fitness =
{alpha_score}")

    print(f"\nBest Fitness: {alpha_score}")
    return alpha_pos


inputs = np.array([
    [0.0, 0.0, 1.0],
    [1.0, 0.0, 1.0],
    [0.0, 1.0, 1.0],
    [1.0, 1.0, 1.0]
])

targets = np.array([0.0, 1.0, 1.0, 0.0])


best_weights = gwo_optimization(inputs, targets, len(inputs))

print("\nEvaluating the final model with the best weights...")
for i in range(len(inputs)):
    output, _ = forward_pass(inputs[i], best_weights)
    print(f"Input: {inputs[i]}, Predicted Output: {output[0]}, Actual
Target: {targets[i]}")
```

Iteration 0/100, Best Fitness = 0.5000013911617378

Iteration 10/100, Best Fitness = 0.5000013911617378

Iteration 20/100, Best Fitness = 0.5000013911617378

Iteration 30/100, Best Fitness = 0.5000013911617378

Iteration 40/100, Best Fitness = 0.5000013911617378

Iteration 50/100, Best Fitness = 0.5000013911617378

Iteration 60/100, Best Fitness = 0.5000013911617378

Iteration 70/100, Best Fitness = 0.5000013911617378

Iteration 80/100, Best Fitness = 0.5000013911617378

Iteration 90/100, Best Fitness = 0.5000013911617378

Best Fitness: 0.5000013911617378

Evaluating the final model with the best weights...

Input: [0. 0. 1.], Predicted Output: -0.0022698934351217197, Actual Target: 0.0

Input: [1. 0. 1.], Predicted Output: -1.0305768090951018e-07, Actual Target: 1.0

Input: [0. 1. 1.], Predicted Output: -1.0305768090951018e-07, Actual Target: 1.0

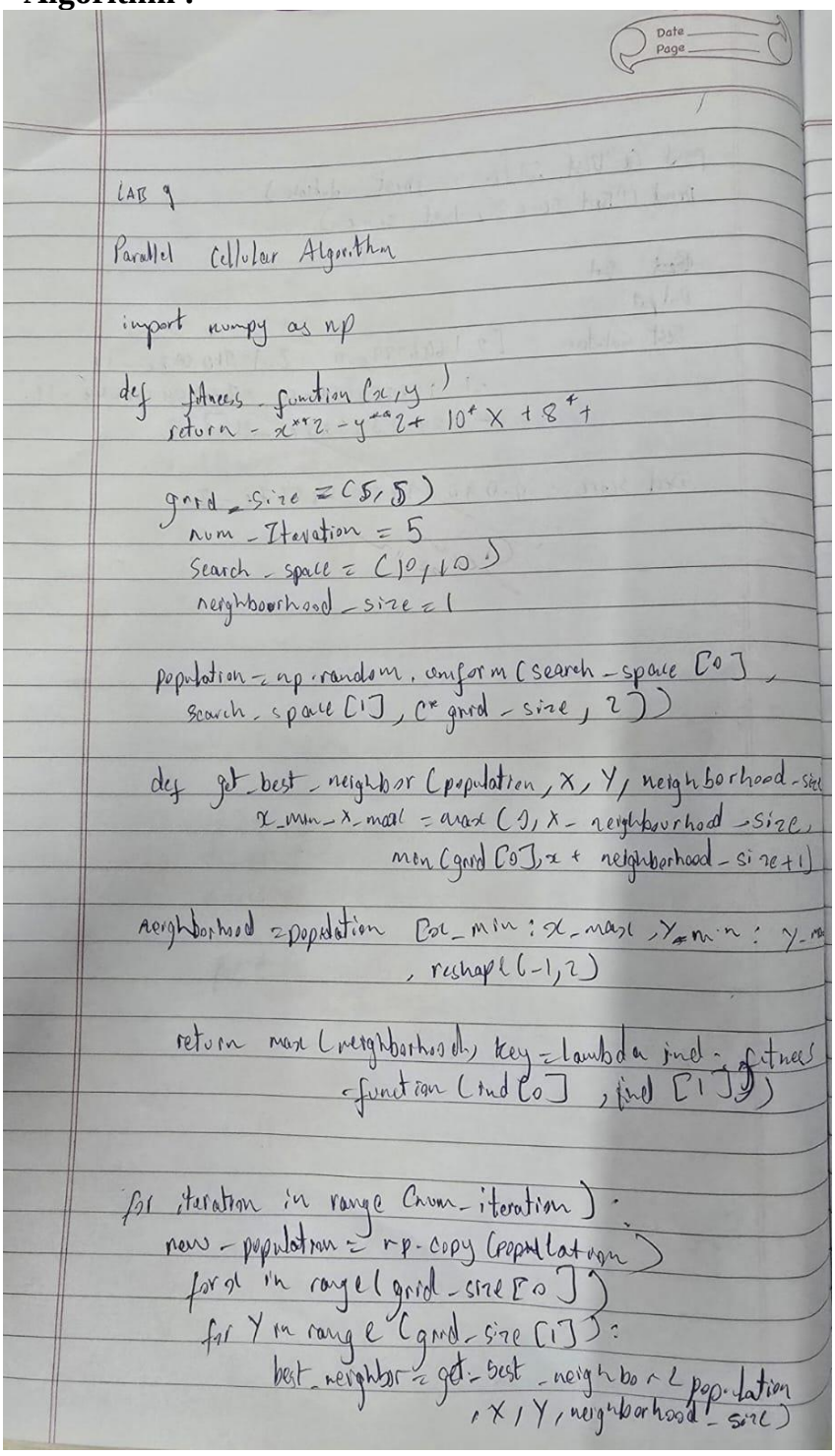
Input: [1. 1. 1.], Predicted Output: -4.678811484419649e-12, Actual Target: 0.0

=== Code Execution Successful ===

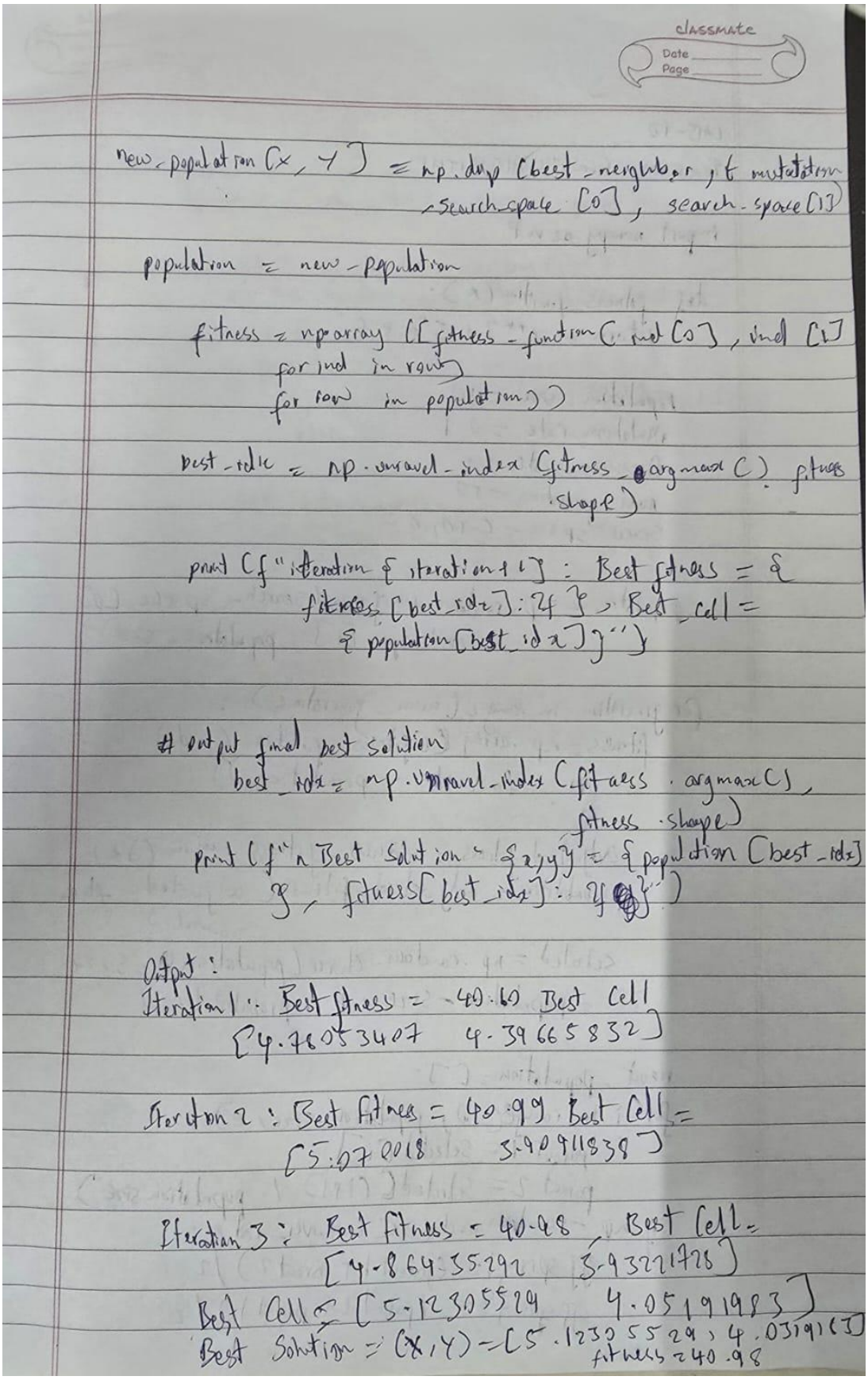
Program 6

Problem Statement : Develop a parallel cellular automaton-based algorithm for optimal robot route planning in a grid-based environment, ensuring collision-free navigation while minimizing travel distance and computational time.

Algorithm :



```
LAB 9

Parallel Cellular Algorithm

import numpy as np

def fitness_function(x,y):
    return -x**2 -y**2+ 10*x +8*y

grid_size = (5,5)
num_Iteration = 5
search_space = (10,10)
neighbourhood_size = 1

population = np.random.uniform(search_space[0],
    search_space[1], (grid_size, 2))

def get_best_neighbor(population, x, y, neighborhood_size
    x_min, x_max = max(0, x- neighbourhood_size,
                   min(grid[0], x + neighborhood_size+1)

neighborhood = population[x_min:x_max, y_min:y_ma
                  , reshape(-1,2)

    return max(neighborhood, key=lambda ind: fitness
        function(ind[0], ind[1]))

for iteration in range(num_iteration):
    new_population = np.copy(population)
    for x in range(grid_size[0]):
        for y in range(grid_size[1]):
            best_neighbor = get_best_neighbor(population
                , x, y, neighborhood_size)
```



```
new_population[x, y] = np.clip(best_neighbor + mutation
                    , search_space[0], search_space[1])

population = new_population

fitness = np.array([[fitness_function(ind[0], ind[1]
          for ind in row]
          for row in population])

best_idx = np.unravel_index(fitness.argmax(), fitness
                    .shape)

print(f"Iteration {iteration+1}: Best fitness = {
      fitness[best_idx]:.2f}, Best_cell =
      {population[best_idx]}")


# output final best solution
    best_idx = np.unravel_index(fitness.argmax(),
                    fitness.shape)
print(f"\n Best Solution: xyyy = {population[best_idx]
      }, fitness[best_idx]:.2f}")
```

Output:

Iteration 1: Best fitness = 40.60 Best Cell
[4.78053407 4.39665832]

Iteration 2: Best fitness = 40.99 Best Cell =
[5.07 0018 3.90911838]

Iteration 3: Best fitness = 40.98 Best Cell =
[4.86435292 3.93221428]

Best Cell = [5.12305529 4.05191983]

Best Solution = (x, y) = [5.12305529, 4.05191983]
fitness = 40.98

Code:

File display

```python
import random
import numpy as np

# Initialize Grid (N x M) with random states (0 or 1)
def initialize_grid(N, M):
    return np.random.choice([0, 1], size=(N, M))

# Count live neighbors for a cell (i, j)
def count_live_neighbors(grid, i, j, N, M):
    return sum(
        grid[ni, nj] for ni in range(i-1, i+2) for nj in range(j-1, j+2)
        if 0 <= ni < N and 0 <= nj < M and (ni != i or nj != j)
    )

# Update cell's state based on neighbors' count
def update_cell(grid, new_grid, i, j, N, M):
    live_neighbors = count_live_neighbors(grid, i, j, N, M)
    if grid[i, j] == 1:
        new_grid[i, j] = 1 if live_neighbors in [2, 3] else 0
    else:
        new_grid[i, j] = 1 if live_neighbors == 3 else 0

# Print the current state of the grid (0s and 1s)
def print_grid(grid):
    for row in grid:
        print(' '.join(map(str, row)))
    print()  # For spacing between generations

# Main Game of Life simulation with printing grid
def parallel_game_of_life(N, M, steps):
    grid = initialize_grid(N, M)

    for _ in range(steps):
        print_grid(grid)  # Print current grid state
        new_grid = np.zeros((N, M), dtype=int)  # New grid for next state
        for i in range(N):
            for j in range(M):
                update_cell(grid, new_grid, i, j, N, M)
        grid = new_grid  # Update grid for next iteration

    print_grid(grid)  # Print final grid after all steps

# Example Usage
N, M = 5, 5  # Smaller grid size for readability
steps = 5  # Number of iterations
final_grid = parallel_game_of_life(N, M, steps)
```

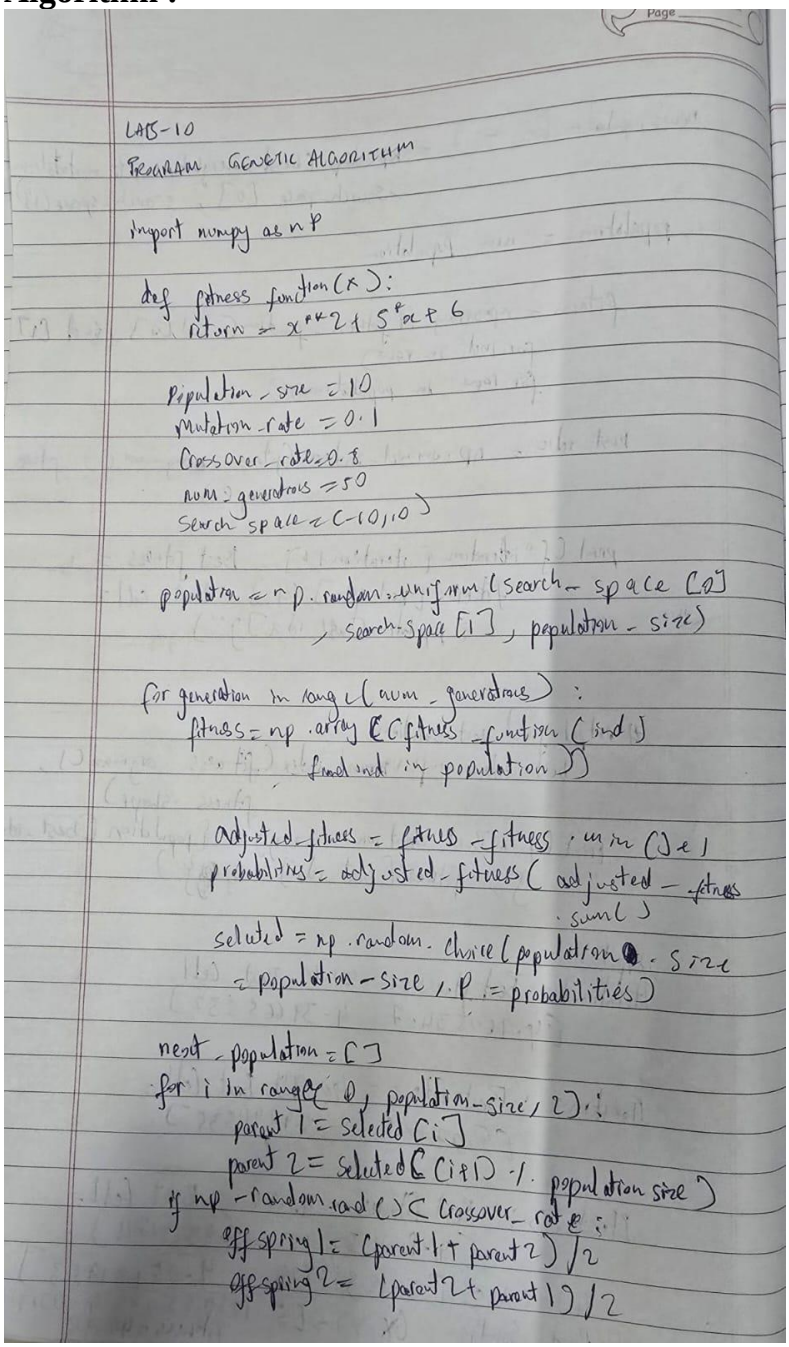
Output:

```
0 0 0 0 0
0 1 0 0 0
0 0 0 1 1
0 1 0 1 1
1 1 1 0 1

0 0 0 0 0
0 0 0 0 0
0 0 0 1 1
1 1 0 0 0
1 1 1 0 1

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
1 0 0 0 1
1 0 1 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 0 0 0
0 1 0 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
```

Program 7

Problem Statement : Solve the 0/1 Knapsack Problem using the Gene Expression Algorithm (GEA) to maximize the total value of selected items without exceeding the given weight capacity.

Algorithm :



```
LAB-10
PROGRAM GENETIC ALGORITHM

import numpy as np

def fitness function(x):
    return = x**2 + 5*x + 6

population_size = 10
mutation_rate = 0.1
crossover_rate = 0.8
num_generations = 50
search_space = (-10,10)

population = np.random.uniform(search_space[0]
    , search_space[1], population_size)

for generation in range(num_generations):
    fitness = np.array([fitness_function(ind)
        for ind in population])

    adjusted_fitness = fitness - fitness.min() + 1
    probabilities = adjusted_fitness / adjusted_fitness
        .sum()
    selected = np.random.choice(population, size
    = population_size, p = probabilities)

next_population = []
for i in range(0, population_size, 2):
    parent1 = selected[i]
    parent2 = selected[(i+1) % population_size]
if np.random.rand() < crossover_rate:
    offspring1 = (parent1 + parent2) / 2
    offspring2 = (parent2 + parent1) / 2
```

```
else:
    offspring1, offspring2 = parent1, parent2

offspring1 = np.random.uniform(-1,1)
    if np.random.rand() < mutation_rate
    else 0
offspring1 = np.random.uniform(1,1)
    if np.random.rand() < Mutation_rate
    else 0

best_idx = fitness.argmax()
print("Generation {generation}:
    Best fitness = fitness[best_idx]:.2f;
    Best Individual = {population[best
    :.2f}")

final_fitness = np.array([fitness_function(ind)
    for ind in population])

best_idx = final_fitness.argmax()
print(f"\n Best solution: X = {population[best_idx]
    :.2f} fitness = {final_fitness[best_idx]:.2f}
```

Output:

Best Solution x = 2.48, fitness = 12.25

Code:

```
import random

# Define the Knapsack Problem (Objective Function)
def knapsack_fitness(items, capacity, solution):
    total_weight = sum([items[i][0] for i in range(len(solution)) if solution[i] == 1])
    total_value = sum([items[i][1] for i in range(len(solution)) if solution[i] == 1])

    # If total weight exceeds the capacity, return 0 (invalid solution)
    if total_weight > capacity:
        return 0
    return total_value

# Gene Expression Algorithm (GEA)
class GeneExpressionAlgorithm:
    def __init__(self, population_size, num_items, mutation_rate, crossover_rate, generations,
capacity, items):
        self.population_size = population_size
        self.num_items = num_items
        self.mutation_rate = mutation_rate
        self.crossover_rate = crossover_rate
        self.generations = generations
        self.capacity = capacity
        self.items = items
        self.population = []

    # Initialize population with random solutions (binary representation)
    def initialize_population(self):
        self.population = [[random.randint(0, 1) for _ in range(self.num_items)] for _ in
range(self.population_size)]

    # Evaluate fitness of the population
    def evaluate_fitness(self):
        return [knapsack_fitness(self.items, self.capacity, individual) for individual in self.population]

    # Select individuals based on fitness (roulette wheel selection)
    def selection(self):
        fitness_values = self.evaluate_fitness()
        total_fitness = sum(fitness_values)
        if total_fitness == 0:  # Avoid division by zero
            return random.choices(self.population, k=self.population_size)
        return random.choices(self.population, weights=[f / total_fitness for f in fitness_values],
k=self.population_size)

    # Crossover (single-point) between two individuals
    def crossover(self, parent1, parent2):
        if random.random() < self.crossover_rate:
```

```
            crossover_point = random.randint(1, self.num_items - 1)
            return parent1[:crossover_point] + parent2[crossover_point:]
        return parent1

    # Mutation (random flip of a gene) of an individual
    def mutation(self, individual):
        if random.random() < self.mutation_rate:
            mutation_point = random.randint(0, self.num_items - 1)
            individual[mutation_point] = 1 - individual[mutation_point]
        return individual

    # Evolve population over generations
    def evolve(self):
        self.initialize_population()
        best_solution = None
        best_fitness = 0

        for gen in range(self.generations):
            # Selection
            selected = self.selection()

            # Crossover and Mutation
            new_population = []
            for i in range(0, self.population_size, 2):
                parent1 = selected[i]
                parent2 = selected[i + 1] if i + 1 < self.population_size else selected[i]

                offspring1 = self.crossover(parent1, parent2)
                offspring2 = self.crossover(parent2, parent1)

                new_population.append(self.mutation(offspring1))
                new_population.append(self.mutation(offspring2))

            self.population = new_population

            # Evaluate fitness and track the best solution
            fitness_values = self.evaluate_fitness()
            max_fitness = max(fitness_values)
            if max_fitness > best_fitness:
                best_fitness = max_fitness
                best_solution = self.population[fitness_values.index(max_fitness)]

            print(f"Generation {gen + 1}: Best Fitness = {best_fitness}")

        return best_solution, best_fitness

# Get user input for the knapsack problem
def get_user_input():
```

```
    print("Enter the number of items:")
    num_items = int(input())
    items = []
    print("Enter the weight and value of each item (space-separated):")
    for i in range(num_items):
        weight, value = map(int, input(f"Item {i + 1}: ").split())
        items.append((weight, value))

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

    return items, capacity, num_items

# Get user input for GEA parameters
def get_algorithm_parameters():
    print("Enter the population size:")
    population_size = int(input())
    print("Enter the mutation rate (e.g., 0.1 for 10%):")
    mutation_rate = float(input())
    print("Enter the crossover rate (e.g., 0.8 for 80%):")
    crossover_rate = float(input())
    print("Enter the number of generations:")
    generations = int(input())

    return population_size, mutation_rate, crossover_rate, generations

# Main function
if __name__ == "__main__":
    # Get user input
    items, capacity, num_items = get_user_input()
    population_size, mutation_rate, crossover_rate, generations = get_algorithm_parameters()

    # Run GEA
    gea = GeneExpressionAlgorithm(population_size, num_items, mutation_rate, crossover_rate,
generations, capacity, items)
    best_solution, best_fitness = gea.evolve()

    print("\nBest Solution:", best_solution)
    print("Best Fitness (Total Value):", best_fitness)
```

```
Enter the number of items:
5
Enter the weight and value of each item (space-separated):
Item 1: 10 20
Item 2: 30 40
Item 3: 50 60
Item 4: 70 80
Item 5: 90 100
Enter the knapsack capacity:
90
Enter the population size:
100
Enter the mutation rate (e.g., 0.1 for 10%):
0.2
Enter the crossover rate (e.g., 0.8 for 80%):
0.9
Enter the number of generations:
10
Generation 1: Best Fitness = 120
Generation 2: Best Fitness = 120
Generation 3: Best Fitness = 120
Generation 4: Best Fitness = 120
Generation 5: Best Fitness = 120
Generation 6: Best Fitness = 120
Generation 7: Best Fitness = 120
Generation 8: Best Fitness = 120
Generation 9: Best Fitness = 120
Generation 10: Best Fitness = 120

Best solution (items selected): [1, 1, 1, 0, 0]
Best fitness (total value): 120
```