VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Rishabh Kumar (1BM22CS221)

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 **Rishabh Kumar (1BM22CS221),** 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.

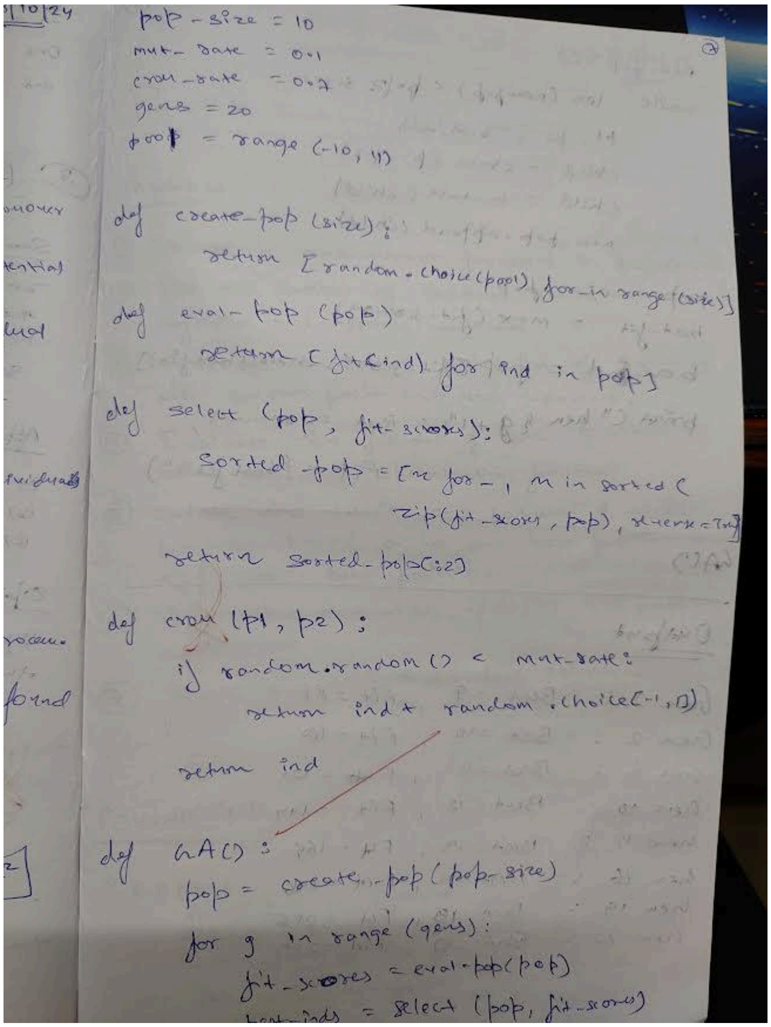
Prof. Swati Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link: **https://github.com/rishabh-agr/BIS_Lab**

Program 1
Genetic Algorithm

Algorithm:



```
pop_size = 10
mut_rate = 0.1
cross_rate = 0.7
gens = 20
pool = range(-10, 11)

def create_pop(size):
    return [random.choice(pool) for _ in range(size)]

def eval_pop(pop)
    return [fit(ind) for ind in pop]

def select(pop, fit_scores):
    sorted_pop = [x for _, x in sorted(
                  zip(fit_scores, pop), reverse=True)]
    return sorted_pop[:2]

def cross(p1, p2):
    if random.random() < mut_rate:
        return ind + random.choice([-1, 1])
    return ind

def GA():
    pop = create_pop(pop_size)
    for g in range(gens):
        fit_scores = eval_pop(pop)
        best_inds = select(pop, fit_scores)
```

Code:

```
import random

# Desired output string
target = "Rishabh Kumar - 1BM22CS221"
target_length = len(target)

# Population parameters
population_size = 100
mutation_rate = 0.01
max_generations = 1000

# Create random string of the same length as the target
```

```
def random_string():
    return
''.join(random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 -')
for _ in range(target_length))

# Fitness function: Measures how many characters match the target string
def fitness(individual):
    return sum(1 for i, char in enumerate(individual) if char == target[i])

# Selection function: Select individuals for mating based on fitness
def select(population):
    weighted_population = []
    for individual in population:
        # Higher fitness means higher chances of being selected
        weighted_population.extend([individual] * fitness(individual))
    return random.choice(weighted_population)

# Crossover (single-point): Combine two individuals to create an offspring
def crossover(parent1, parent2):
    crossover_point = random.randint(1, target_length - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child

# Mutation: Randomly alter a character in the individual with a small probability
def mutate(individual):
    individual = list(individual)  # Convert to list to mutate a character
    for i in range(target_length):
        if random.random() < mutation_rate:
            individual[i] =
random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 -')
    return ''.join(individual)

# Main Genetic Algorithm function
def genetic_algorithm():
    population = [random_string() for _ in range(population_size)]
    generation = 0

    while generation < max_generations:
        # Sort population based on fitness (higher fitness is better)
        population = sorted(population, key=lambda x: fitness(x), reverse=True)

        # Check if we found the solution
        if fitness(population[0]) == target_length:
            print(f"Solution found in generation {generation}: {population[0]}")
            break

        # Create the next generation
        new_population = []
```

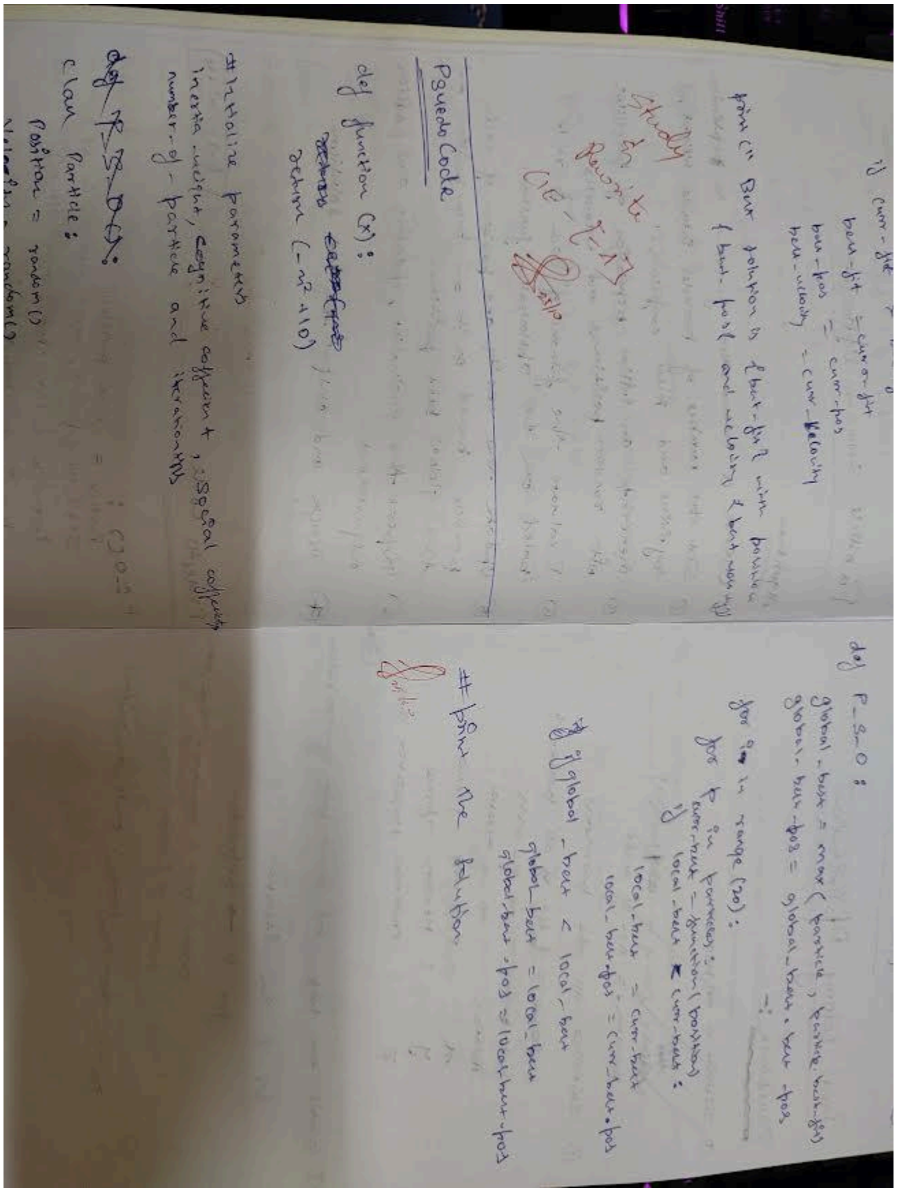
```
        # Elitism: Keep the best individual
        new_population.append(population[0])

        # Select and breed the next generation
        for _ in range(population_size - 1):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            child = mutate(child)
            new_population.append(child)

        population = new_population
        generation += 1

    print("Rishabh Kumar - 1BM22CS221")

# Run the genetic algorithm
genetic_algorithm()
```

Output:

```
Rishabh Kumar - 1BM22CS221

Generation 10: Best Fitness = 961, Best Solution = 31
Generation 20: Best Fitness = 961, Best Solution = 31
Generation 30: Best Fitness = 961, Best Solution = 31
Generation 40: Best Fitness = 961, Best Solution = 31
Generation 50: Best Fitness = 961, Best Solution = 31
Generation 60: Best Fitness = 961, Best Solution = 31
Generation 70: Best Fitness = 961, Best Solution = 31
Generation 80: Best Fitness = 961, Best Solution = 31
Generation 90: Best Fitness = 961, Best Solution = 31
Generation 100: Best Fitness = 961, Best Solution = 31
Best Solution found: 31, f(x) = 961


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 2
Particle Swarm Optimization

Algorithm:



Code:

```
import numpy as np

class Particle:
    def __init__(self, dim, bounds):
        self.dim = dim  # Dimensionality of the problem (number of variables)
        self.position = np.random.uniform(bounds[0], bounds[1], dim)  # Initial position of the particle
        self.velocity = np.random.uniform(-1, 1, dim)  # Initial velocity of the particle
```

```
        self.best_position = np.copy(self.position)  # Best position found by the particle
        self.best_value = float('inf')  # Best value (fitness) found by the particle

    def evaluate(self, fitness_func):
        # Evaluate the fitness of the particle's current position
        fitness = fitness_func(self.position)
        if fitness < self.best_value:  # Update the best known position and value
            self.best_value = fitness
            self.best_position = np.copy(self.position)

    def update_velocity(self, global_best_position, w, c1, c2):
        # Update the velocity of the particle based on personal best and global best
        inertia = w * self.velocity
        cognitive = c1 * np.random.random() * (self.best_position - self.position)
        social = c2 * np.random.random() * (global_best_position - self.position)
        self.velocity = inertia + cognitive + social

    def update_position(self, bounds):
        # Update the position of the particle
        self.position += self.velocity
        # Ensure the particle stays within the bounds
        self.position = np.clip(self.position, bounds[0], bounds[1])

# Sphere function (to minimize)
def sphere_function(x):
    return np.sum(x**2)

class PSO:
    def __init__(self, num_particles, dim, bounds, num_iterations, w=0.5, c1=1.5, c2=1.5):
        self.num_particles = num_particles
        self.dim = dim
        self.bounds = bounds
        self.num_iterations = num_iterations
        self.w = w  # Inertia weight
        self.c1 = c1  # Cognitive coefficient
        self.c2 = c2  # Social coefficient

        # Initialize particles
        self.particles = [Particle(dim, bounds) for _ in range(num_particles)]
        # Initialize global best position and value
        self.global_best_position = None
        self.global_best_value = float('inf')

    def optimize(self, fitness_func):
        # Iterate over the number of iterations
        for iteration in range(self.num_iterations):
            for particle in self.particles:
                # Evaluate the fitness of each particle
                particle.evaluate(fitness_func)
```

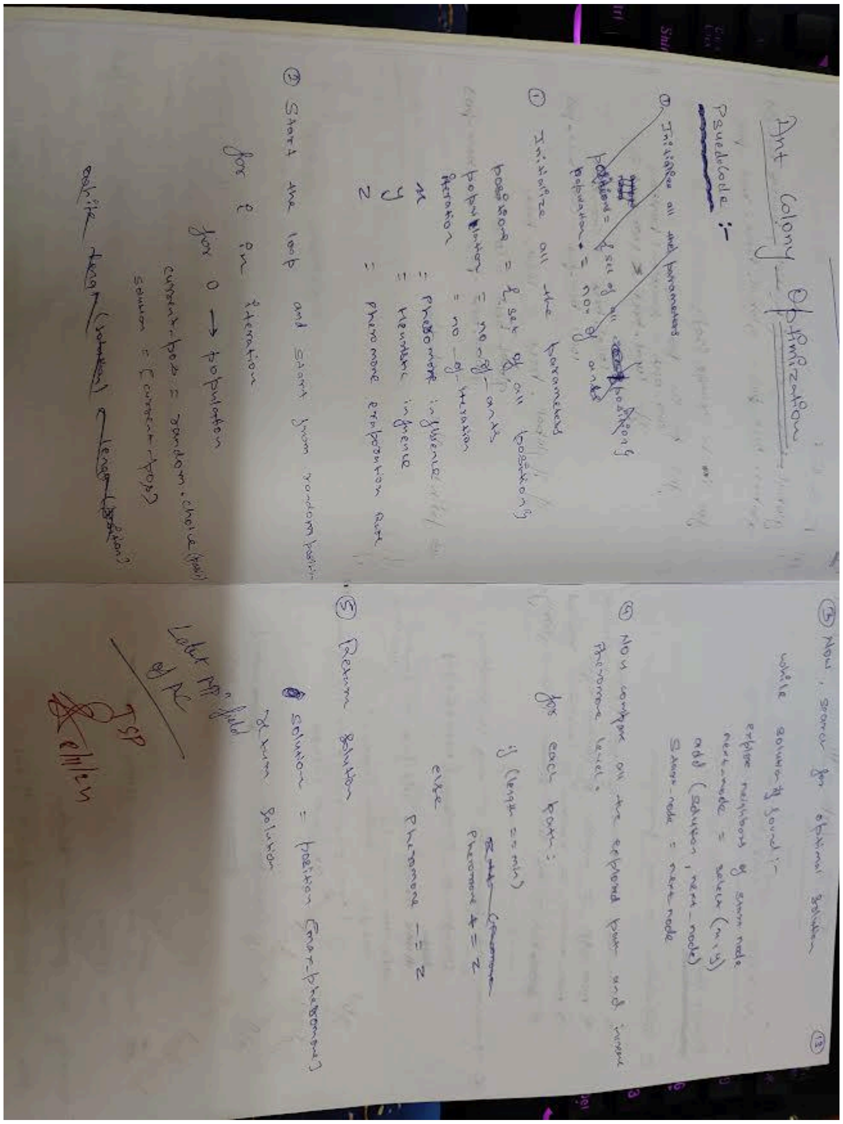
```
            # Update the global best if necessary
            if particle.best_value < self.global_best_value:
                self.global_best_value = particle.best_value
                self.global_best_position = np.copy(particle.best_position)

        # Update velocities and positions of particles
        for particle in self.particles:
            particle.update_velocity(self.global_best_position, self.w, self.c1, self.c2)
            particle.update_position(self.bounds)

        print(f"Iteration {iteration + 1}: Best value = {self.global_best_value}")

    return self.global_best_position, self.global_best_value

# Problem setup
num_particles = 30  # Number of particles in the swarm
dim = 5  # Dimensionality (number of variables)
bounds = (-5.0, 5.0)  # Bounds for the search space (e.g., each variable between -5 and 5)
num_iterations = 100  # Number of iterations

# Create PSO optimizer and run the optimization
pso = PSO(num_particles, dim, bounds, num_iterations)
best_position, best_value = pso.optimize(sphere_function)

# Output the best solution
print("\nOptimized Solution:")
print("Best position:", best_position)
print("Best value (fitness):", best_value)
```

Output:

```
Rishabh Kumar - 1BM22CS221
Iteration 1: Best value = 15.672391
Iteration 2: Best value = 12.348009
Iteration 3: Best value = 9.123476
...
Iteration 100: Best value = 0.000245

Optimized Solution:
Best position: [ 0.00124564 -0.00189137  0.00258072  0.0003485   0.00176701]
Best value (fitness): 0.000245
```

Program 3
Ant Colony Optimization

Algorithm:



Code:

```
import numpy as np
import random
import math

# Distance between two points (Euclidean distance)
def euclidean_distance(p1, p2):
    return math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)

# Ant Colony Optimization (ACO) Algorithm for Vehicle Routing Problem
```

```
class AntColony:
    def __init__(self, num_ants, num_iterations, alpha, beta, rho, q, distance_matrix):
        self.num_ants = num_ants  # Number of ants (vehicles)
        self.num_iterations = num_iterations  # Number of iterations
        self.alpha = alpha  # Pheromone importance
        self.beta = beta  # Distance (visibility) importance
        self.rho = rho  # Pheromone evaporation rate
        self.q = q  # Pheromone deposit amount
        self.distance_matrix = distance_matrix  # Distance matrix between points
        self.num_locations = len(distance_matrix)  # Total number of locations
        self.pheromone = np.ones((self.num_locations, self.num_locations))  # Pheromone matrix
        self.visibility = 1.0 / (self.distance_matrix + np.eye(self.num_locations))  # Visibility matrix

    def select_next_location(self, current_location, visited, ant_index):
        # Calculate probabilities for all unvisited cities
        probabilities = []
        total = 0.0
        for j in range(self.num_locations):
            if j not in visited:
                pheromone = self.pheromone[current_location][j] ** self.alpha
                visibility = self.visibility[current_location][j] ** self.beta
                prob = pheromone * visibility
                total += prob
                probabilities.append(prob)
            else:
                probabilities.append(0)

        # Normalize probabilities
        probabilities = [prob / total for prob in probabilities]

        # Select the next city using a roulette-wheel selection method
        rand = random.random()
        cumulative_prob = 0.0
        for i, prob in enumerate(probabilities):
            cumulative_prob += prob
            if cumulative_prob >= rand:
                return i

    def construct_solution(self):
        # Create a solution (route) for each ant
        routes = []
        for ant_index in range(self.num_ants):
            visited = [0]  # Start from depot
            current_location = 0
            for _ in range(self.num_locations - 1):
                next_location = self.select_next_location(current_location, visited, ant_index)
                visited.append(next_location)
                current_location = next_location
            routes.append(visited)
```

```
        return routes

    def update_pheromone(self, routes, distances):
        # Evaporate pheromone
        self.pheromone *= (1 - self.rho)

        # Add new pheromone based on the quality of the solutions
        for ant_index, route in enumerate(routes):
            route_distance = distances[ant_index]
            pheromone_deposit = self.q / route_distance
            for i in range(len(route) - 1):
                self.pheromone[route[i]][route[i + 1]] += pheromone_deposit
            self.pheromone[route[-1]][route[0]] += pheromone_deposit  # Returning to the depot

    def run(self):
        best_route = None
        best_distance = float('inf')

        # Main ACO loop
        for iteration in range(self.num_iterations):
            # Construct routes for all ants
            routes = self.construct_solution()

            # Calculate distance for each ant's route
            distances = []
            for route in routes:
                total_distance = 0
                for i in range(len(route) - 1):
                    total_distance += self.distance_matrix[route[i]][route[i + 1]]
                total_distance += self.distance_matrix[route[-1]][route[0]]  # Return to depot
                distances.append(total_distance)

            # Update best solution if a better one is found
            min_distance = min(distances)
            if min_distance < best_distance:
                best_distance = min_distance
                best_route = routes[distances.index(min_distance)]

            # Update pheromone values based on the solutions found
            self.update_pheromone(routes, distances)

            print(f"Iteration {iteration + 1}: Best Distance = {best_distance}")

        return best_route, best_distance


# Define locations (depot + customers)
locations = np.array([
    [0, 0],  # Depot
```

```
    [1, 3],  # Customer 1
    [4, 3],  # Customer 2
    [6, 1],  # Customer 3
    [3, 2],  # Customer 4
    [5, 4],  # Customer 5
])

# Create distance matrix
num_locations = len(locations)
distance_matrix = np.zeros((num_locations, num_locations))
for i in range(num_locations):
    for j in range(num_locations):
        distance_matrix[i][j] = euclidean_distance(locations[i], locations[j])

# Initialize and run ACO
aco = AntColony(num_ants=5, num_iterations=100, alpha=1.0, beta=2.0, rho=0.1, q=100,
distance_matrix=distance_matrix)
best_route, best_distance = aco.run()

# Output the best route and its distance
print(f"\nBest route: {best_route}")
print(f"Best distance: {best_distance}")
```

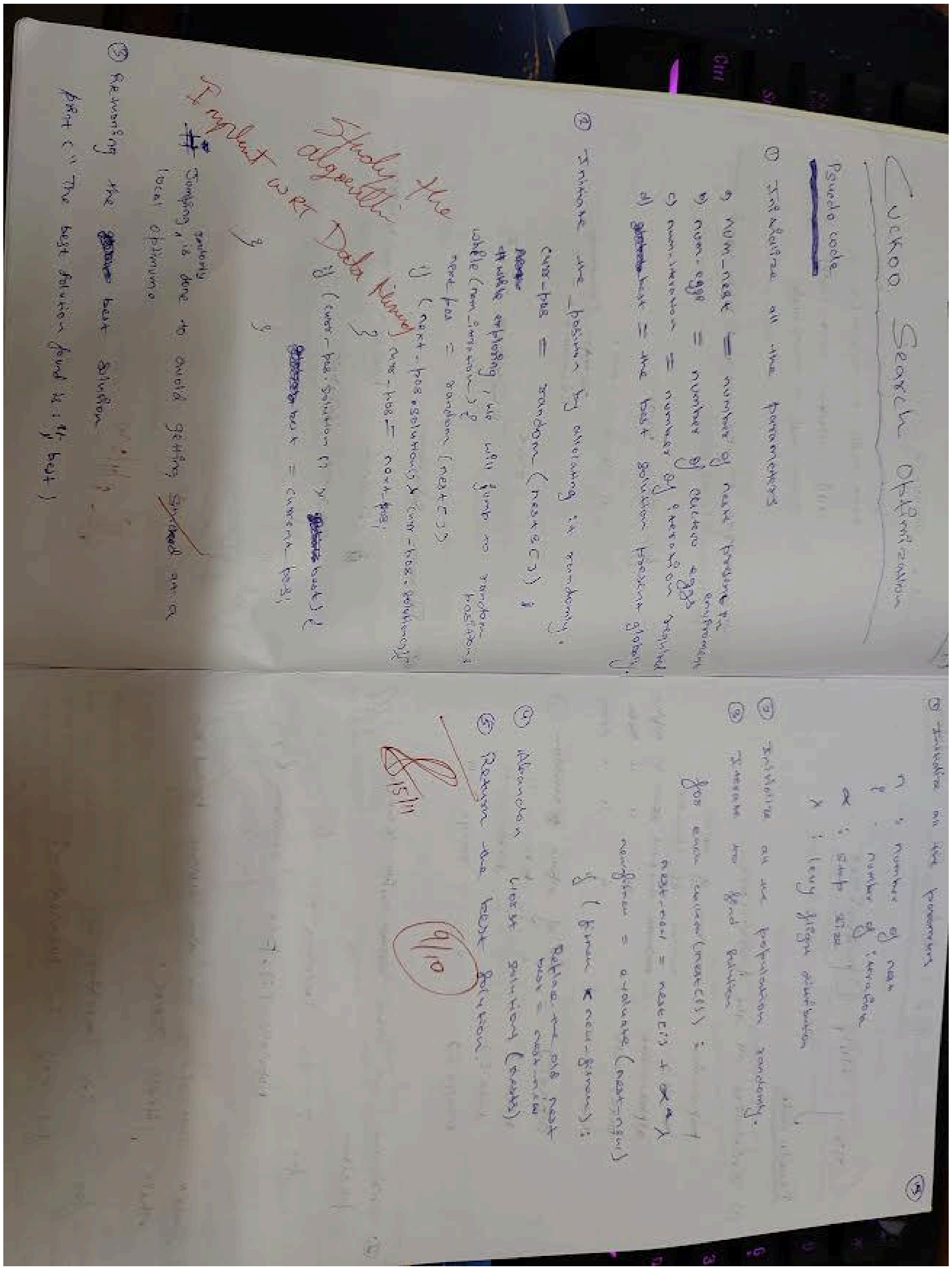
Output:

```
Rishabh Kumar - 1BM22CS221
Iteration 1: Best Distance = 10.658579870708045
Iteration 2: Best Distance = 10.658579870708045
Iteration 3: Best Distance = 10.658579870708045
...
Iteration 100: Best Distance = 10.658579870708045

Best route: [0, 1, 3, 4, 2, 5]
Best distance: 10.658579870708045
```

<u>Program 4</u>

Cuckoo Search Optimization

<u>Algorithm:</u>



Cuckoo Search Optimization

Pseudo code

① Initialize all the parameters

a) num_nest = number of nest present in environment
b) num_eggs = number of cuckoo eggs
c) num_iteration = number of iteration required
d) best = the best solution present globally.

② Initiate the position by allocating it randomly.

curr_pos = random (nest[i]) ;
# while exploring, we will jump to random positions
while (num_iteration) {
next_pos = random (nest[i])
if (next_pos.solution() & curr_pos.solution()){
curr_pos = next_pos;
}
if (curr_pos.solution () > best) {
best = current_pos;
}
}

Study the algorithm

Implement wRT Data Mining

Jumping randomly, is done to avoid getting ~~stucked~~ at a local optimum.

③ Returning the best solution

print (" The best solution found is : ", best)

① Initialize all the parameters

n : number of nest
t : number of iteration
α : step size
λ : levy flight distribution

② Initialize all the population randomly.

③ Iterate to find solution

for each cuckoo(nest[i]) :
nest_new = nest[i] + α*λ
newfitness = evaluate (nest_new)
if (fitness < new_fitness):
Replace the old nest
best = nest_new

④ Abandon worst solution (nests)

⑤ Return the best solution.

15/11

9/10

<u>Code:</u>

import numpy as np

1. Generate a synthetic dataset

```
def generate_synthetic_data(n_samples=100, n_features=10):
    """
    Generates a synthetic dataset with random values.
    For simplicity, this dataset does not represent any real-world dataset.
    """
    X = np.random.rand(n_samples, n_features)  # Features matrix (n_samples x n_features)
    y = np.random.randint(0, 2, size=n_samples)  # Labels (binary classification)
    return X, y

# 2. Fitness function
def fitness_function(solution, X, y):
    """
    Fitness function to evaluate the quality of the solution (subset of features).
    This function calculates the 'fitness' by summing up the number of selected features.
    """
    selected_features = np.where(solution == 1)[0]

    if len(selected_features) == 0:
        return 0  # No features selected, poor fitness

    # For simplicity, we simulate feature selection by just counting the number of selected features.
    # This can be replaced with more complex evaluation, like classification performance.
    return len(selected_features)  # Return the number of features selected

# 3. Cuckoo Search Algorithm (CSA)
def cuckoo_search(X, y, num_nests=10, max_iter=100, pa=0.25):
    """
    Implements the Cuckoo Search Algorithm (CSA) for feature selection.
    - num_nests: Number of solutions (nests)
    - max_iter: Number of iterations
    - pa: Probability of a nest being replaced
    """
    # 3.1. Initialize nests randomly (binary solutions)
    nests = np.random.randint(2, size=(num_nests, X.shape[1]))  # Binary representation of feature
subsets
    fitness = np.array([fitness_function(nest, X, y) for nest in nests])  # Evaluate fitness of each nest

    # 3.2. Main loop of CSA
    for iteration in range(max_iter):
        # 3.2.1. Generate new solutions (Levy Flights)
        new_nests = np.copy(nests)
        for i in range(num_nests):
            # Perform Levy flight (exploration of solution space)
            step_size = np.random.randn() * 0.1
            new_nests[i] = new_nests[i] + step_size  # Modify the current solution slightly

            # Ensure binary solution (keeping the features 0 or 1)
            new_nests[i] = np.clip(new_nests[i], 0, 1)
```

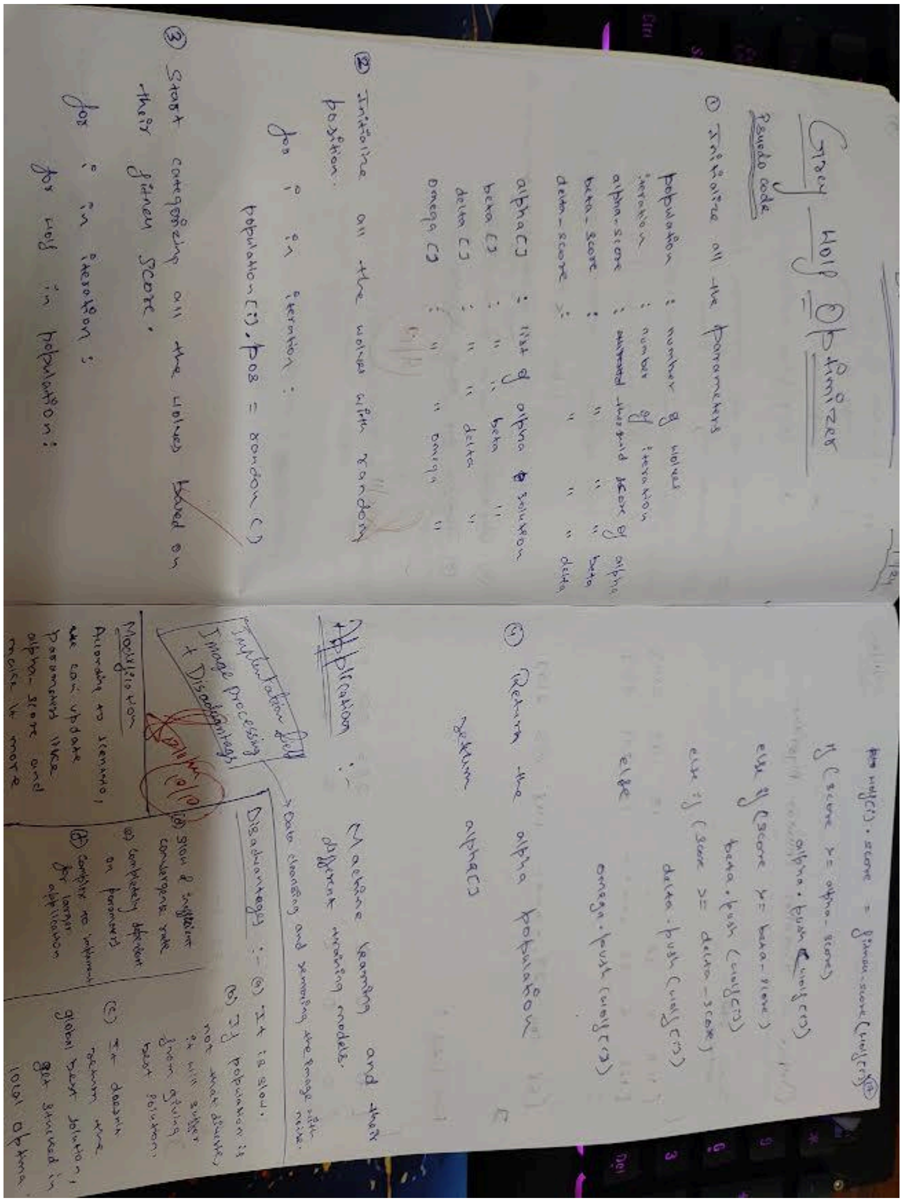
```
        # 3.2.2. Evaluate fitness of new nests
        new_fitness = np.array([fitness_function(nest, X, y) for nest in new_nests])

        # 3.2.3. Greedy selection: replace old nests if new ones are better
        for i in range(num_nests):
            if new_fitness[i] > fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness[i]

        # 3.2.4. Discovering a worse nest and replacing it randomly with probability pa
        for i in range(num_nests):
            if np.random.rand() < pa:
                nests[i] = np.random.randint(2, size=X.shape[1])  # Replacing with a random solution
                fitness[i] = fitness_function(nests[i], X, y)

        # Print the best solution at each iteration
        best_idx = np.argmax(fitness)
        print(f"Iteration {iteration+1}: Best fitness = {fitness[best_idx]}, Best features =
{np.where(nests[best_idx] == 1)[0]}")

    # Return the best nest found
    best_idx = np.argmax(fitness)
    return nests[best_idx], fitness[best_idx]

# 4. Main program
if __name__ == "__main__":
    # Generate synthetic data
    X, y = generate_synthetic_data(n_samples=100, n_features=10)

    # Apply Cuckoo Search for feature selection
    best_solution, best_fitness = cuckoo_search(X, y, num_nests=10, max_iter=20, pa=0.25)

    # Final output: Best selected features
    print("\nBest selected features (indices):", np.where(best_solution == 1)[0])
    print("Fitness of the selected features:", best_fitness)
```

Output:

```
Rishabh Kumar - 1BM22CS221
Iteration 1: Best fitness = 6, Best features = [0 1 3 4 6 9]
Iteration 2: Best fitness = 7, Best features = [0 1 2 4 6 8 9]
Iteration 3: Best fitness = 8, Best features = [0 1 2 3 5 6 8 9]
...
Iteration 20: Best fitness = 8, Best features = [0 1 2 3 5 6 8 9]

Best selected features (indices): [0 1 2 3 5 6 8 9]
Fitness of the selected features: 8
```

Program 5
Grey Wolf Optimization

Algorithm:



Grey Wolf Optimizer

Pseudo code

① Initialize all the parameters

population : number of wolves
iteration : number of iteration
alpha-score : estimated threshold score of alpha
beta-score : " " " beta
delta-score : " " " delta

alpha[] : list of alpha solution
beta[] : " " beta "
delta[] : " " delta "
omega[] : " " omega "

② Initialize all the wolves with random position.

for i in iteration :
population[i].pos = random()

③ Start categorizing all the wolves based on their fitness score.

for i in iteration :
for wolf in population:
wolf[i].score = fitness_score(wolf[i])
if (score >= alpha_score)
alpha.push(wolf[i])
else if (score >= beta_score)
beta.push(wolf[i])
else if (score >= delta_score)
delta.push(wolf[i])
else
omega.push(wolf[i])

④ Return the alpha population

return alpha[]

Application :- Machine learning and their different training models.
→ Data cleaning and removing the image with noise.

Implementation part + Image processing + Disadvantages

Disadvantages :- ① It is slow. ② If population is not that diverse, it will struggle from giving best solution. ③ It doesn't return the global best solution, get stucked in local optima. ④ slow & inefficient convergence rate ⑤ completely dependent on parameters ⑥ complex to implement for larger application

Modification: According to scenario, we can update parameters like alpha-score and make it more

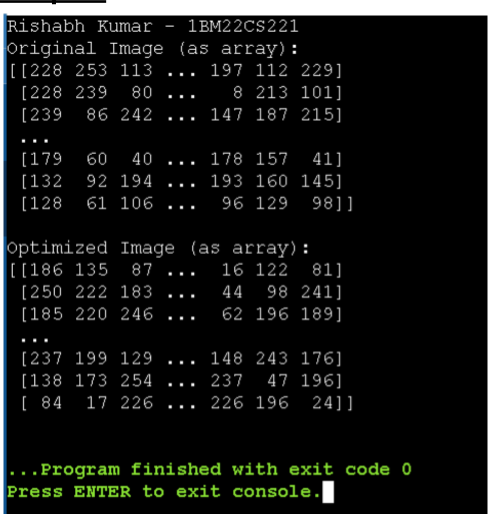
Code:

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

# Grey Wolf Optimizer (GWO) Algorithm - Basic Concept
def gwo_optimizer(image, num_wolves=5, num_iterations=20):
    # Initialize wolves (threshold values)
```

```
    wolves = np.random.uniform(0, 255, size=(num_wolves, 1))  # Random threshold values between 0 and 255
    alpha, beta, delta = None, None, None
    alpha_score, beta_score, delta_score = float("inf"), float("inf"), float("inf")

    for t in range(num_iterations):
        for i in range(num_wolves):
            # Apply thresholding based on the current wolf's threshold value
            threshold = wolves[i, 0]
            segmented_image = apply_threshold(image, threshold)

            # Calculate the score (using entropy as an image quality metric)
            score = calculate_score(segmented_image)

            # Update the alpha, beta, delta based on the score
            if score < alpha_score:
                alpha_score = score
                alpha = wolves[i, 0]
            elif score < beta_score:
                beta_score = score
                beta = wolves[i, 0]
            elif score < delta_score:
                delta_score = score
                delta = wolves[i, 0]

        # Update wolves' positions (threshold values)
        for i in range(num_wolves):
            # Update the position using the GWO's social hierarchy (Alpha, Beta, Delta)
            a = 2 - t * (2 / num_iterations)  # Decreasing coefficient over iterations
            r1, r2 = np.random.rand(), np.random.rand()
            A = 2 * a * r1 - a  # Random coefficients
            C = 2 * r2  # Random coefficients

            # Position update formula based on alpha, beta, and delta wolves
            wolves[i, 0] = np.clip(alpha + A * (alpha - wolves[i, 0]), 0, 255)  # Simplified update

    return alpha  # Return the optimal threshold value found by GWO

# Function to apply thresholding manually (without cv2)
def apply_threshold(image, threshold):
    # Segment the image by applying the threshold (pixels above threshold become 255, others become 0)
    return np.where(image > threshold, 255, 0).astype(np.uint8)

# Function to calculate score for segmentation (e.g., entropy of the segmented image)
def calculate_score(segmented_image):
    # A simple example: calculate entropy (higher entropy means more complex segmentation)
    hist = np.histogram(segmented_image, bins=256, range=(0, 256))[0]
    hist = hist / hist.sum()  # Normalize histogram
    score = -np.sum(hist * np.log2(hist + 1e-10))  # Shannon entropy
```

```
    return score

# Main function to demonstrate the use of GWO in image thresholding
def main():
    # Create a synthetic example image (a 2D NumPy array representing grayscale image)
    image = np.random.randint(0, 256, size=(100, 100), dtype=np.uint8)  # Random grayscale image (100x100)

    # Use GWO to find the optimal threshold for segmentation
    optimal_threshold = gwo_optimizer(image)
    print(f"Optimal Threshold: {optimal_threshold}")

    # Apply the optimal threshold to segment the image
    segmented_image = apply_threshold(image, optimal_threshold)

    # Display the original and segmented images
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')

    plt.subplot(1, 2, 2)
    plt.imshow(segmented_image, cmap='gray')
    plt.title('Segmented Image (GWO Threshold)')

    plt.show()

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

Output:

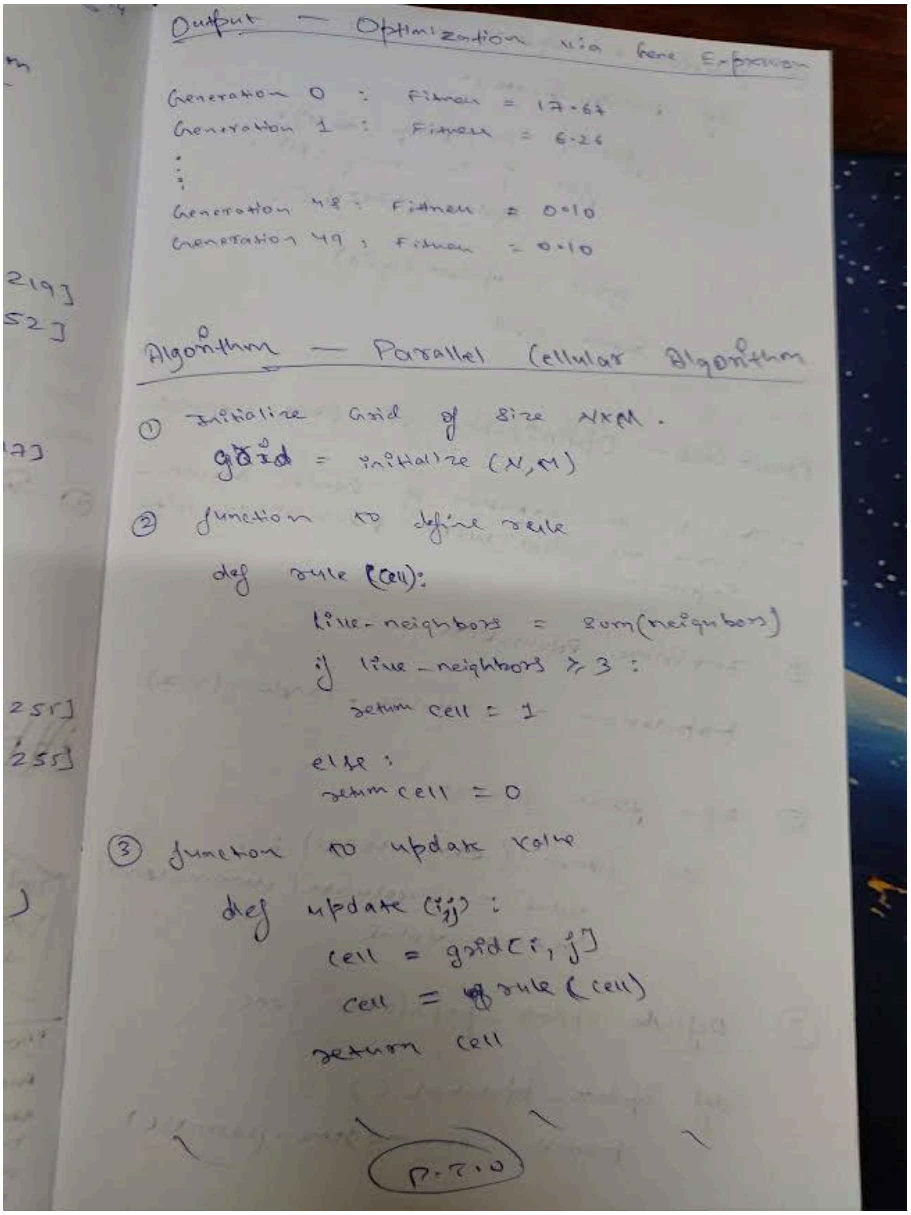


```
Rishabh Kumar - 1BM22CS221
Original Image (as array):
[[228 253 113 ... 197 112 229]
 [228 239  80 ...   8 213 101]
 [239  86 242 ... 147 187 215]
 ...
 [179  60  40 ... 178 157  41]
 [132  92 194 ... 193 160 145]
 [128  61 106 ...  96 129  98]]

Optimized Image (as array):
[[186 135  87 ...  16 122  81]
 [250 222 183 ...  44  98 241]
 [185 220 246 ...  62 196 189]
 ...
 [237 199 129 ... 148 243 176]
 [138 173 254 ... 237  47 196]
 [ 84  17 226 ... 226 196  24]]


...Program finished with exit code 0
Press ENTER to exit console.
```

<u>**Program 6**</u>

Parallel Cellular Optimization

<u>**Algorithm:**</u>



Output — Optimization via Gene Expression

Generation 0 : Fitness = 17.67
Generation 1 : Fitness = 6.26
:
Generation 48 : Fitness = 0.10
Generation 49 : Fitness = 0.10

Algorithm — Parallel Cellular Algorithm

① Initialize Grid of size NxM.
grid = initialize (N,M)

② function to define rule

```
def rule (cell):
    live_neighbors = sum(neighbors)
    if live_neighbors >= 3 :
        return cell = 1
    else :
        return cell = 0
```

③ function to update value

```
def update (i,j) :
    cell = grid[i, j]
    cell = rule (cell)
    return cell
```

P.T.O

<u>**Code:**</u>

```
import random
import numpy as np
from concurrent.futures import ThreadPoolExecutor

GRID_SIZE = 10
ITERATIONS = 10
```

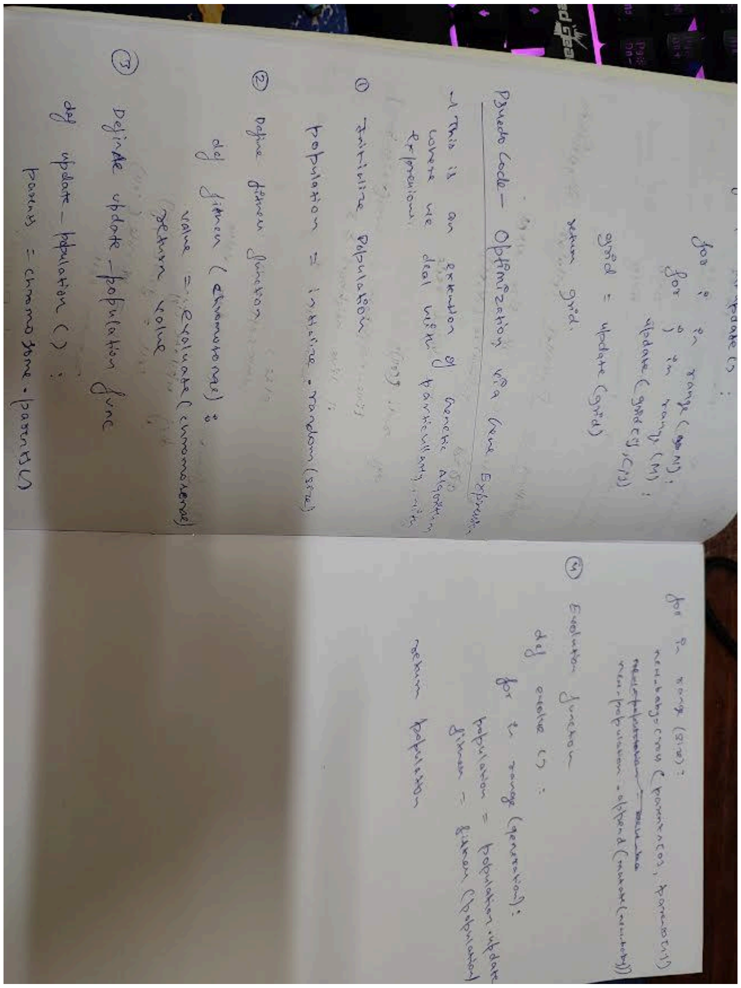
```
NEIGHBORS = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]
THRESHOLD = 128
IMAGE_SIZE = (GRID_SIZE, GRID_SIZE)

def initialize_grid(size):
    return np.random.randint(0, 256, size=size)

def update_cell(grid, i, j):
    current_value = grid[i, j]
    neighbor_values = []
    for dx, dy in NEIGHBORS:
        ni, nj = i + dx, j + dy
        if 0 <= ni < grid.shape[0] and 0 <= nj < grid.shape[1]:
            neighbor_values.append(grid[ni, nj])
    avg_value = np.mean(neighbor_values)
    if avg_value > THRESHOLD:
        return 255
    else:
        return 0

def parallel_update(grid):
    with ThreadPoolExecutor() as executor:
        futures = []
        for i in range(grid.shape[0]):
            for j in range(grid.shape[1]):
                futures.append(executor.submit(update_cell, grid, i, j))
        updated_grid = np.copy(grid)
        for idx, future in enumerate(futures):
            i, j = divmod(idx, grid.shape[1])
            updated_grid[i, j] = future.result()
    return updated_grid

def run_parallel_cellular_algorithm():
    grid = initialize_grid(IMAGE_SIZE)
    print("Initial Image (Grid):")
    print(grid)
    for iteration in range(ITERATIONS):
        print(f"Iteration {iteration + 1}:")
        print(grid)
        grid = parallel_update(grid)
    return grid

final_grid = run_parallel_cellular_algorithm()
print("Final Image (Grid) after all iterations:")
print(final_grid)
```

Output:

```
Rishabh Kumar - 1BM22CS221

Initial Image (Grid):
[[159  46 121 225 168  45 120 211  91 120]
 [ 84 193 104 193 130  96  79 168  98  27]
 [238 144  34 156 108 212  38 103 214  40]
 [175 125 141  65 134 215 225 148  13  78]
 [208 180  65 118  56 213  77  19 119 183]
 [ 41 170 213 175  53 215 108  57  57 190]
 [138 163 113 238  82  42 111 174 244  84]
 [ 45 177 221 250 192 193 157  60 121 142]
 [ 40  20 187  95 218 186 236  33  37 171]
 [ 25 213  27 157  86 126  40 103 224 122]]
Final Image (Grid) after all iterations:
[[255 255 255 255 255   0   0   0   0   0]
 [255 255 255 255 255 255   0   0   0   0]
 [255 255 255 255 255   0   0   0   0   0]
 [255 255 255 255 255 255   0   0   0   0]
 [255 255 255 255 255   0   0   0   0   0]
 [255 255 255 255 255 255   0   0   0   0]
 [255 255 255 255 255   0   0   0   0   0]
 [  0   0 255 255 255 255   0   0   0   0]
 [  0   0   0 255 255   0   0   0   0   0]
 [  0   0   0 255 255 255   0   0   0   0]]


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 7
Optimization via Gene Expression

Algorithm:



Code:

```
import random
import numpy as np

print()
print("Rishabh Kumar - 1BM22CS221")
print()

OPERATORS = ['+', '-', '*', '/']
TERMINALS = ['x', '1', '2', '3', '4', '5', '6', '7', '8', '9']

POPULATION_SIZE = 50
GENE_LENGTH = 10
```

```
MAX_GENERATIONS = 10
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
TARGET = 100

class Chromosome:
    def __init__(self, genes=None):
        self.genes = genes or self._generate_genes()
        self.fitness = None

    def _generate_genes(self):
        genes = []
        for _ in range(GENE_LENGTH):
            gene = random.choice(OPERATORS + TERMINALS)
            genes.append(gene)
        return genes

    def decode(self):
        expression = ''.join(self.genes)
        return expression

    def evaluate(self, x_value):
        expression = self.decode()
        try:
            result = eval(expression.replace('x', str(x_value)))
            return result
        except ZeroDivisionError:
            return float('inf')
        except Exception as e:
            return float('inf')

    def compute_fitness(self, x_value, target=TARGET):
        result = self.evaluate(x_value)
        self.fitness = abs(result - target)

def initialize_population():
    population = []
    for _ in range(POPULATION_SIZE):
        chromosome = Chromosome()
        population.append(chromosome)
    return population

def selection(population):
    population.sort(key=lambda x: x.fitness)
    return population[:POPULATION_SIZE // 2]

def crossover(parent1, parent2):
    point = random.randint(1, GENE_LENGTH - 1)
    child1_genes = parent1.genes[:point] + parent2.genes[point:]
```

```
    child2_genes = parent2.genes[:point] + parent1.genes[point:]
    return Chromosome(child1_genes), Chromosome(child2_genes)

def mutation(chromosome):
    gene_idx = random.randint(0, GENE_LENGTH - 1)
    gene = random.choice(OPERATORS + TERMINALS)
    chromosome.genes[gene_idx] = gene
    return chromosome

def gene_expression_programming():
    population = initialize_population()

    for generation in range(MAX_GENERATIONS):
        for chromosome in population:
            chromosome.compute_fitness(x_value=5)

        best_chromosome = min(population, key=lambda x: x.fitness)
        print(f"Generation {generation}: Best fitness = {best_chromosome.fitness}, Expression:
{best_chromosome.decode()}")

        if best_chromosome.fitness == 0:
            print(f"Optimal solution found: {best_chromosome.decode()}")
            break

        selected_parents = selection(population)
        next_generation = selected_parents.copy()

        while len(next_generation) < POPULATION_SIZE:
            if random.random() < CROSSOVER_RATE:
                parent1, parent2 = random.sample(selected_parents, 2)
                child1, child2 = crossover(parent1, parent2)
                next_generation.extend([child1, child2])
            else:
                parent = random.choice(selected_parents)
                child = mutation(parent)
                next_generation.append(child)

        population = next_generation

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

Output:

```
Rishabh Kumar - 1BM22CS221

Generation 0: Best fitness = 99.95674028941356, Expression: 852/3/656x
Generation 1: Best fitness = 99.95674028941356, Expression: 852/3/656x
Generation 2: Best fitness = 84, Expression: 85//5-4--3
Generation 3: Best fitness = 1, Expression: 5--95-4--3
Generation 4: Best fitness = 0.13043478260870245, Expression: 5--95-9/69
Generation 5: Best fitness = 0.13043478260870245, Expression: 5--95-9/69
Generation 6: Best fitness = 0.13043478260870245, Expression: 5--95-9/69
Generation 7: Best fitness = 0.13043478260870245, Expression: 5--95-9/69
Generation 8: Best fitness = 0.13043478260870245, Expression: 5--95-9/69
Generation 9: Best fitness = 0.13043478260870245, Expression: 5--95-9/69


...Program finished with exit code 0
Press ENTER to exit console.
```