VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shruti Khandelia (1BM22CS274)

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 **Shruti Khandelia (1BM22CS274),** 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.

Spoorthi D M Assistant Professor Department of CSE, BMSCE	Dr. Jyothi S Nayak Professor & HOD Department of CSE, BMSCE

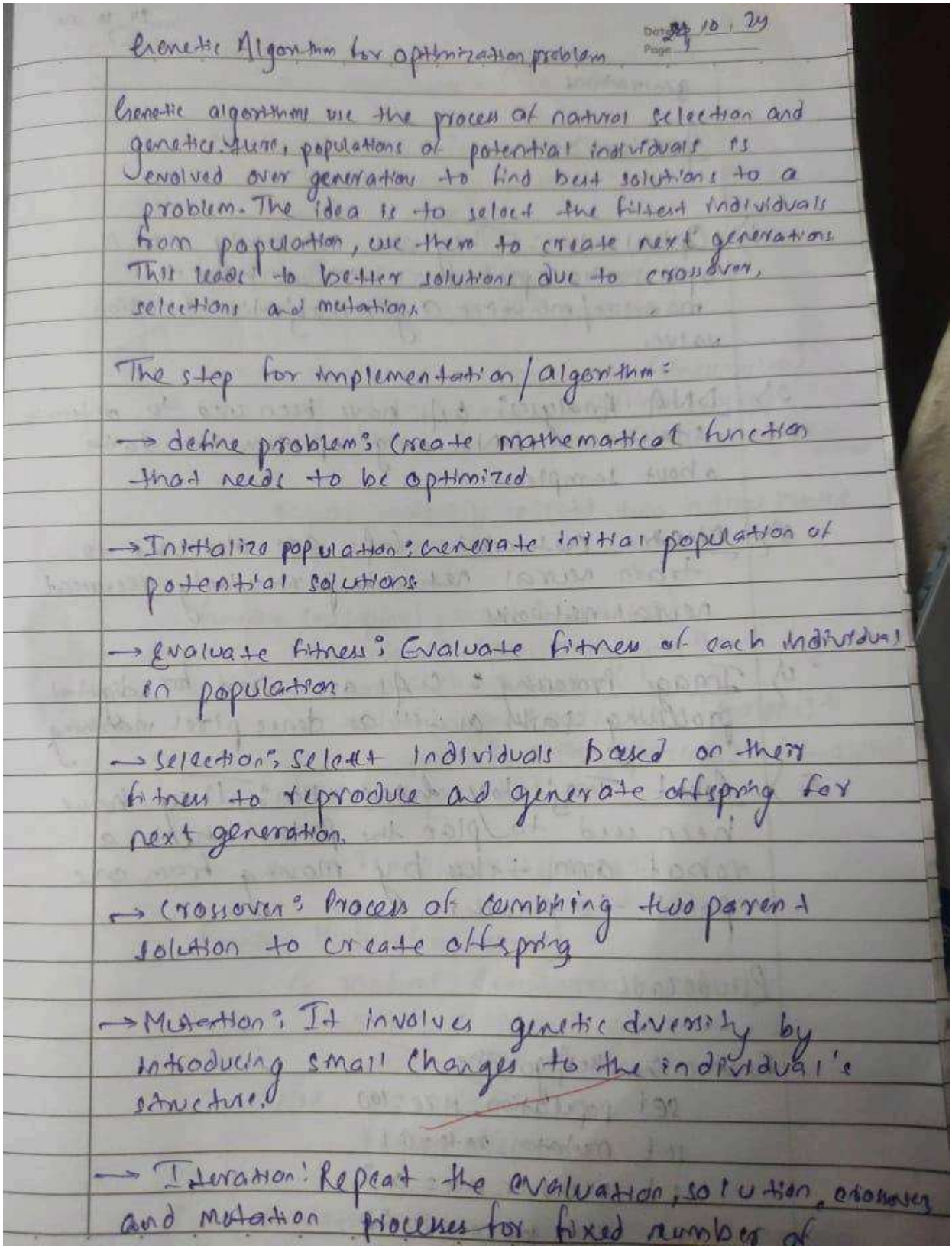
Index

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

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:



Genetic Algorithm for optimization problem

Date 24/10/24
Page

Genetic algorithms use the process of natural selection and genetics. Here, populations of potential individuals is evolved over generation to find best solutions to a problem. The idea is to select the fittest individuals from population, use them to create next generations. This leads to better solutions due to crossover, selections and mutations.

The step for implementation / algorithm:

→ define problem: Create mathematical function that needs to be optimized

→ Initialize population: Generate initial population of potential solutions.

→ Evaluate fitness: Evaluate fitness of each individual in population.

→ Selection: Select individuals based on their fitness to reproduce and generate offspring for next generation.

→ Crossover: Process of combining two parent solution to create offspring

→ Mutation: It involves genetic diversity by introducing small changes to the individual's structure.

→ Iteration: Repeat the evaluation, solution, crossover and mutation processes for fixed number of

Applications:

1) **Optimization:** They are commonly used in optimization problem where we have to maximize/minimize a given objective function value.

2) **DNA Analysis:** GAs have been used to determine structure of DNA using spectrometric data about sample.

3) **Neural Networks:** GAs are also used to train Neural networks, particularly recurrent Neural networks.

4) **Image Processing:** GAs are used for digital processing tasks as well as dense pixel matching.

5) **Robot Trajectory Generation:** They have been used to plan the paths which a robot arm takes by moving from one point to another.

Pseudocode

```
Function GeneticAlgorithm
    set population_size = 100
    set mutation_rate = 0.1
    set crossover_rate = 0.7
    set num_generations = 50
```

Date 24/10/24
Page 9

```
let value_range = (-10, 10)

Function InitializePopulation(size, range)
    return randomly generate 'size' individuals
    within 'range'

Function EvaluateFitness(population)
    return [f(x) for x in population]

Function SelectParent(population, fitness_values)
    let total_fitness = sum(fitness_values)
    let selection_probabilities = fitness_values/total_fitness

    return randomly selected two indices based
    on probabilities

function Crossover(parent1, parent2)
    if random() < crossover_rate then
        let alpha = Random()
        return (alpha * parent1 + (1-alpha) *
            parent2, alpha * parent2 + (1-alpha)
            * parent1)
    else
        return (parent1, parent2)

function Mutate(offspring)
    if random() < mutation_rate then
        return randomly generated a new
        value within value range
    else
        return offspring

let population = InitializePopulation(population_size,
                                      value_range)
```

for generation from 1 to num_generations
do
set fitness_values = Evaluate Fitness
(Population)
set best_index = Index of max(fitness_values)
print "Generation", generation,
"Best solution = ", population(best_index)
", fitness = ", fitness_values[best_index]

Repeat
set new_population = []

for i from 1 to (population_size/2) do
set (parent1, parent2) = (selectParent
(population, fitness_values)
set (offspring1, offspring2) = (crossover
(parent1, parent2)
Add Mutate(offspring1) to new_population
Add Mutate(offspring2) to new_population

Set population = new_population

Set fitness_values = Evaluate Fitness (population)
set best_index = Index of max(fitness_values)
Print "final Best solution = ", population[best_index], "
fitness = ", fitness_values[best_index]

END function

24/10/24

Code:

```
import random

# Set a random seed for reproducibility
random.seed(42)

def fitness(chromosome):
    x = int(''.join(map(str, chromosome)), 2)
    return x ** 2

def binary_string_to_chromosome(binary_string):
    return [int(bit) for bit in binary_string]

def generate_population_from_input():
    population = []
    for _ in range(population_size):
        while True:
            binary_string = input("Enter a binary string of size 5 (e.g., '11001'): ")
            if len(binary_string) == 5 and all(bit in '01' for bit in binary_string):
                population.append(binary_string_to_chromosome(binary_string))
                break
            else:
                print("Invalid input. Please enter a binary string of size 5.")
    return population

def select_pair(population, fitnesses):
    total_fitness = sum(fitnesses)
    selection_probs = [f / total_fitness for f in fitnesses]
    parent1 = population[random.choices(range(len(population)), selection_probs)[0]]
    parent2 = population[random.choices(range(len(population)), selection_probs)[0]]
    return parent1, parent2

def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring2

def mutate(chromosome, mutation_rate):
    return [gene if random.random() > mutation_rate else 1 - gene for gene in chromosome]
```

```
# Parameters
population_size = 4
generations = 20
mutation_rate = 0.01

# Initialize population from user input
population = generate_population_from_input()

for generation in range(generations):
    fitnesses = [fitness(chromosome) for chromosome in population]

    new_population = []

    # Create new population
    while len(new_population) < population_size:
        parent1, parent2 = select_pair(population, fitnesses)
        offspring1, offspring2 = crossover(parent1, parent2)
        new_population.append(mutate(offspring1, mutation_rate))
        new_population.append(mutate(offspring2, mutation_rate))

    # Ensure the new population has the right size
    population = new_population[:population_size]

# Get the maximum fitness
fitnesses = [fitness(chromosome) for chromosome in population]
max_fitness = max(fitnesses)

print(f"Maximum Possible Fitness: {max_fitness}")
```

Output:

```
Enter a binary string of size 5 (e.g., '11001'): 11011
Enter a binary string of size 5 (e.g., '11001'): 01011
Enter a binary string of size 5 (e.g., '11001'): 11100
Enter a binary string of size 5 (e.g., '11001'): 01101
Maximum Possible Fitness: 841
```

Program 2

Ant Colony Optimization

Algorithm:

Date 14, 11, 24
Page 7

3) Ant Colony Optimization for Travelling Salesman Problem

Initialize parameters
n_cities, alpha, beta, rho, Q, iterations, ant_count, initial_pheromone

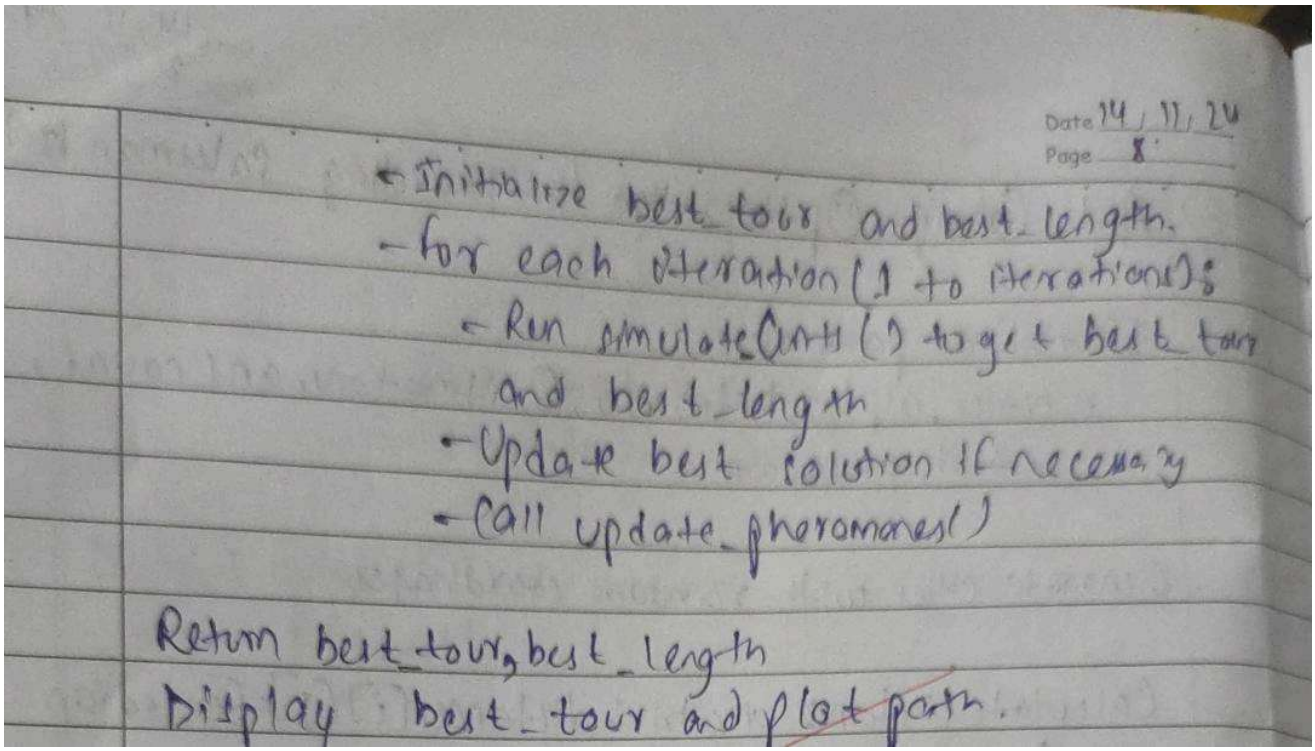
Generate cities with random coordinates.

Calculate distance matrix: distances[i][j] = Euclidian distance between cities i and j

Initialize pheromone matrix: pheromone[i][j] = initial_pheromone

Define functions:
- calc_distance(city1, city2): Return Euclidian distance
- choose_next_city(current, visited, pheromone, distances):
  - Compute probability for each unvisited city
  - Return city with highest probability.
- simulate_ants(pheromone, distances):
  - for each ant, construct a tour by selecting cities based on pheromone and distance.
  - Calculate tour length and update best tour if shorter
  - Return best_tour and best_length
- update_pheromones(pheromone, all_tours, best_tour, best_length, Q, rho):
  - Evaporate pheromones: pheromone *= (1-rho)
  - Deposit pheromone on all tours based on tour quality
  - Reinforce pheromone on best tour

Main loop (ACO algorithm):


Date 14/11/24
Page 8

- Initialize best_tour and best_length.
- for each iteration (1 to iterations):
  - Run simulateAnt() to get best_tour and best_length
  - Update best solution if necessary
  - Call update_pheromones()

Return best_tour, best_length
Display best_tour and plot path.

Pseudocode

```
initialize parameters
    n_cities = 10
    alpha = 1.0
    beta = 5.0
    rho = 0.5
    Q = 100
    iterations = 100
    ant count = 50
    initial pheromone = 1.0

np.random.seed(42)
cities = np.random.rand(n_cities, 2)

function calc_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0])**2 +
                   (city[1] - city[1]**2)

distances = np.zeros((n_cities, n_cities))
for i in range(n_cities):
    for j in range(i+1, n_cities):
        dist = calc_distance(cities[i], cities[j])
        distances[i][j] = dist
        distances[j][i] = dist

pheromone = np.ones((n_cities, n_cities)) * initial_pheromone

function choose_next_city(current_city, visited_city, pheromone,
        probability=[] distance)
    total = 0.0
    for i in range(n_cities):
        if i not in visited:
```

```
pheromone_level = pheromone[current_city][i] ** alpha
distance_heuristic = (1.0 / distances[current_city][i]) ** beta
prob = pheromone_level * distance_heuristic
probabilities.append(prob)
else:
    probabilities.append(0)

return random.choices(range(n_cities), probabilities)[0]

function simulate_ants(pheromone, distances):
    for _ in range(ant_count):
        tour = [random.randint(0, n_cities-1)]
        visited.add(tour[0])

    while len(visited) < n_cities:
        current_city = tour[-1]
        tour.append(next_city)
        visited.add(next_city)

    tour_length = sum(distances[tour[i]][tour[i+1]]
                      for i in range(n_cities-1)) +
                  distances[tour[-1]][tour[0]]

    if tour_length < best_length:
        best_length = tour_length
        return best_tour, best_length, all_tours

function update_pheromones(pheromone, all_tours,
                           best_tours, best_length, Q, rho):
```

```
pheromone *= (1 - rho)
for tour, length in all_tours:
    for i in range(len(tour) - 1):
        pheromone[tour[i]][tour[i+1]] += Q / length
for i in range(len(best_tour) - 1):
    pheromone[best_tour[i]][best_tour[i+1]] += Q / best_length
    pheromone[best_tour[i+1]][best_tour[i]] += Q / best_length.

function aco_tsp():
    for iteration in range(iterations):
        if (best_length < best_overall_length):
            best_overall_length = best_length
            best_overall_tour = best_tour
        update_pheromones(pheromone, all_tours,
        best_tours, best_length, Q, rho)
    return best_overall_tour, best_overall_length

best_tour, best_length = aco_tsp()
x = [cities[city][0] for city in best_tour] +
        [cities[best_tour[0]][0]]
y = [cities[city][1] for city in best_tour] + [cities[best_t
        [0][1]]
plt.figure(figsize = (8,6))
plt.plot(x, y, marker = 'o', color = 'b', linestyle = '-', marker=
```

~~best_tour, best_length = aco_tsp()~~

two
21/1/24

Code:

```
import random
import numpy as np
import operator

FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]

def decode_chromosome(chromosome, x):
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output

def fitness_function(chromosome, target_function, x_values):
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
    return mse


def selection(population, fitnesses):
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
```

```
    return population[np.random.choice(len(population), p=probabilities)]

def mutate(chromosome, mutation_rate=0.1):
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome

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

def ant_colony_optimization(cost_matrix, n_ants=10, n_iterations=100, evaporation_rate=0.5,
alpha=1, beta=2):
    n_nodes = len(cost_matrix)
    pheromones = np.ones((n_nodes, n_nodes))  # Initialize pheromones

    def calculate_probability(i, j, visited):
        if j in visited:
            return 0
        return (pheromones[i][j] ** alpha) * ((1 / cost_matrix[i][j]) ** beta)

    def construct_solution():
        path = [random.randint(0, n_nodes - 1)]
        while len(path) < n_nodes:
            i = path[-1]
            probabilities = [calculate_probability(i, j, path) for j in range(n_nodes)]
            total = sum(probabilities)
            probabilities = [p / total if total > 0 else 0 for p in probabilities]
            next_node = np.random.choice(range(n_nodes), p=probabilities)
            path.append(next_node)
        path.append(path[0])  # Return to start
        return path

    def path_cost(path):
        return sum(cost_matrix[path[i]][path[i + 1]] for i in range(len(path) - 1))

    best_path = None
    best_cost = float('inf')
```

```
    for iteration in range(n_iterations):
        solutions = [construct_solution() for _ in range(n_ants)]
        costs = [path_cost(solution) for solution in solutions]
        for i, cost in enumerate(costs):
            if cost < best_cost:
                best_cost = cost
                best_path = solutions[i]

        pheromones *= (1 - evaporation_rate)  # Evaporation
        for i, solution in enumerate(solutions):
            for j in range(len(solution) - 1):
                pheromones[solution[j]][solution[j + 1]] += 1 / costs[i]

        print(f"Iteration {iteration + 1}: Best Cost = {best_cost}")

    print("Best Path:", best_path)
    print("Best Cost:", best_cost)

cost_matrix = [
    [0, 2, 2, 5, 7],
    [2, 0, 4, 8, 2],
    [2, 4, 0, 1, 3],
    [5, 8, 1, 0, 2],
    [7, 2, 3, 2, 0]
]
ant_colony_optimization(cost_matrix, n_ants=5, n_iterations=20)
```

Output:

```
Iteration 15: Best Cost = 9
Iteration 16: Best Cost = 9
Iteration 17: Best Cost = 9
Iteration 18: Best Cost = 9
Iteration 19: Best Cost = 9
Iteration 20: Best Cost = 9
Best Path: [1, 0, 2, 3, 4, 1]
Best Cost: 9
```

Program 3

Particle Swarm Optimization

Algorithm:

Particle Swarm Optimization

Initialize swarm
for each particle i in swarm:
initialize position (x_i) randomly within problem bounds
initialize velocity (v_i) randomly
set personal best position pbest_i = x_i
set personal best value f(pbest_i)

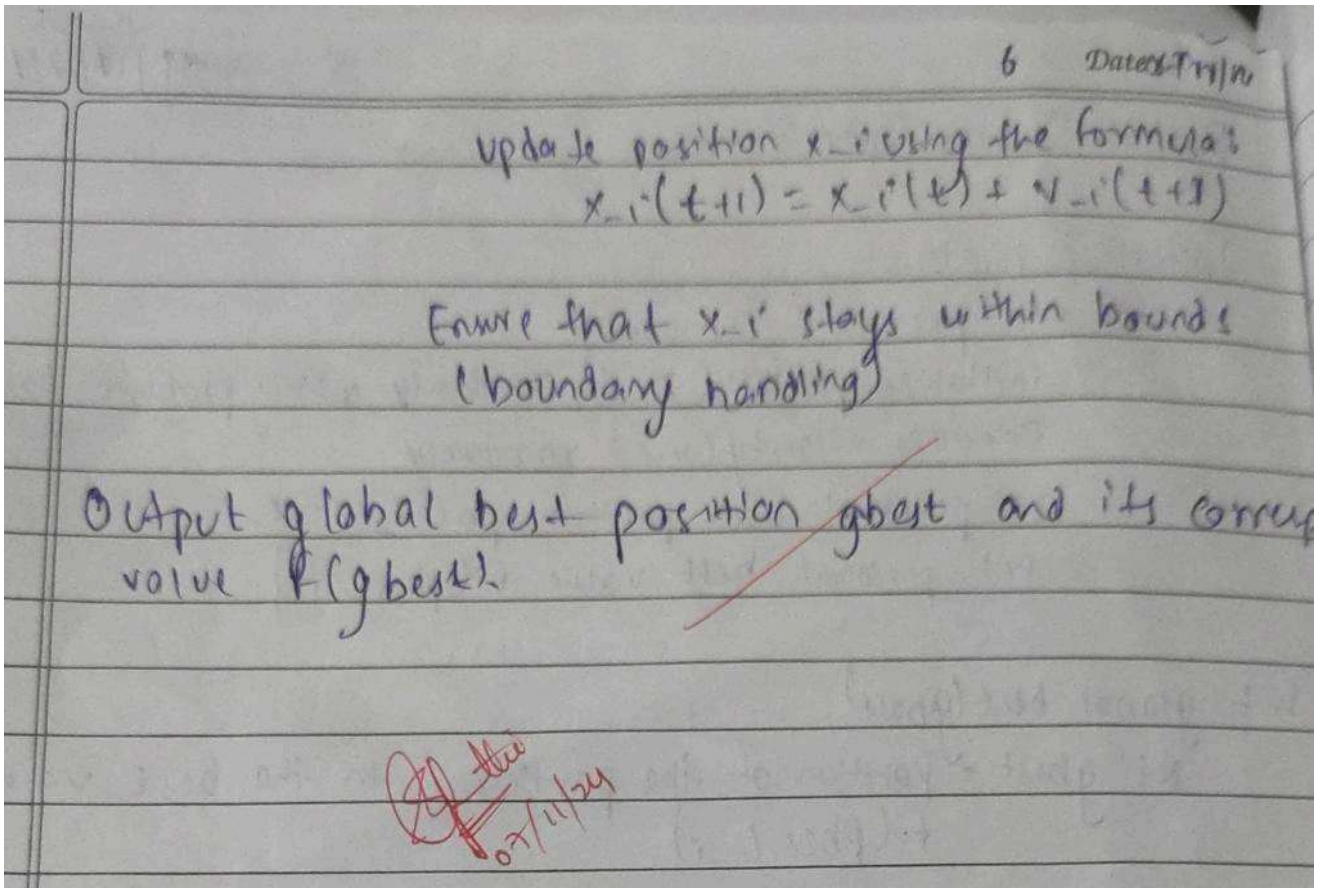
set global best (gbest)
set gbest = position of the particle with the best value f(pbest_i).

for each iteration (t = 1 to max_iterations):
for each particle i:
Evaluate fitness f(x_i) at current position x_i
if f(x_i) < f(pbest_i):
update pbest_i = x_i
update f(pbest_i) = f(x_i)
if f(pbest_i) < f(gbest):
update gbest = pbest_i

for each particle i:
update velocity v_i using the formula:
$v_i(t+1) = w * v_i(t) + c1 * r1 * (pbest_i - x_i) + c2 * r2 * (gbest - x_i)$

where:
w = inertia weight (controls exploration v/s exploitation)
c1, c2 = cognitive and social learning factors
r1, r2 = random values between 0 and 1.
pbest_i = personal best position of particle i
gbest = global best position of the swarm.

for each particle i:



Code:

```
import random
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import animation

def fitness_function(x1, x2):
    f1 = x1 + 2 * -x2 + 3
    f2 = 2 * x1 + x2 - 8
    z = f1**2 + f2**2
    return z

def update_velocity(particle, velocity, pbest, gbest, w_min=0.5, max=1.0, c=0.1):
    new_velocity = np.zeros_like(particle)
    r1 = random.uniform(0, max)
    r2 = random.uniform(0, max)
    w = random.uniform(w_min, max)

    for i in range(len(particle)):
```

```
        new_velocity[i] = (w * velocity[i] +
                           c * r1 * (pbest[i] - particle[i]) +
                           c * r2 * (gbest[i] - particle[i]))
    return new_velocity

def update_position(particle, velocity):
    new_particle = particle + velocity
    return new_particle

def pso_2d(population, dimension, position_min, position_max, generation, fitness_criterion):
    # Initialization
    particles = np.array([[random.uniform(position_min, position_max) for _ in range(dimension)] for
_ in range(population)])
    pbest_position = particles.copy()
    pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])

    gbest_index = np.argmin(pbest_fitness)
    gbest_position = pbest_position[gbest_index]

    velocity = np.zeros((population, dimension))

    images = []  # For animation

    for t in range(generation):
        if np.average(pbest_fitness) <= fitness_criterion:
            break

        for n in range(population):
            velocity[n] = update_velocity(particles[n], velocity[n], pbest_position[n], gbest_position)
            particles[n] = update_position(particles[n], velocity[n])

        pbest_fitness = np.array([fitness_function(p[0], p[1]) for p in particles])
        for n in range(population):
            if pbest_fitness[n] < fitness_function(pbest_position[n][0], pbest_position[n][1]):
                pbest_position[n] = particles[n]

        gbest_index = np.argmin(pbest_fitness)
        gbest_position = pbest_position[gbest_index]

        # Plotting the current positions of the particles
        fig = plt.figure(figsize=(10, 10))
        ax = fig.add_subplot(111, projection='3d')
```

```
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')

    x = np.linspace(position_min, position_max, 80)
    y = np.linspace(position_min, position_max, 80)
    X, Y = np.meshgrid(x, y)
    Z = fitness_function(X, Y)
    ax.plot_wireframe(X, Y, Z, color='r', linewidth=0.2)

    ax.scatter3D(
        particles[:, 0],
        particles[:, 1],
        [fitness_function(p[0], p[1]) for p in particles],
        c='b'
    )

    # Capture the frame for animation
    plt.title(f'Generation: {t + 1}')
    plt.tight_layout()
    plt.savefig(f'frame_{t}.png')
    plt.close(fig)

# Create animation
frames = [plt.imread(f'frame_{i}.png') for i in range(t)]
fig, ax = plt.subplots(figsize=(10, 10))
ax.axis('off')
image = ax.imshow(frames[0])

def update(frame):
    image.set_array(frames[frame])
    return image,

ani = animation.FuncAnimation(fig, update, frames=len(frames), interval=100)
ani.save('./pso_simple.gif', writer='pillow')

# Print the results
print('Global Best Position: ', gbest_position)
print('Best Fitness Value: ', min(pbest_fitness))
print('Average Particle Best Fitness Value: ', np.average(pbest_fitness))
print('Number of Generations: ', t)
```

# Run the PSO algorithm
pso_2d(population=30, dimension=2, position_min=-10, position_max=10, generation=100, fitness_criterion=1e-3)

Output:

```
Global Best Position:  [2.59992843 2.79914636]
Best Fitness Value:  3.6691186243893878e-06
Average Particle Best Fitness Value:  0.0007223322365523365
Number of Generations:  45
```

Program 4

Cuckoo Search Algorithm

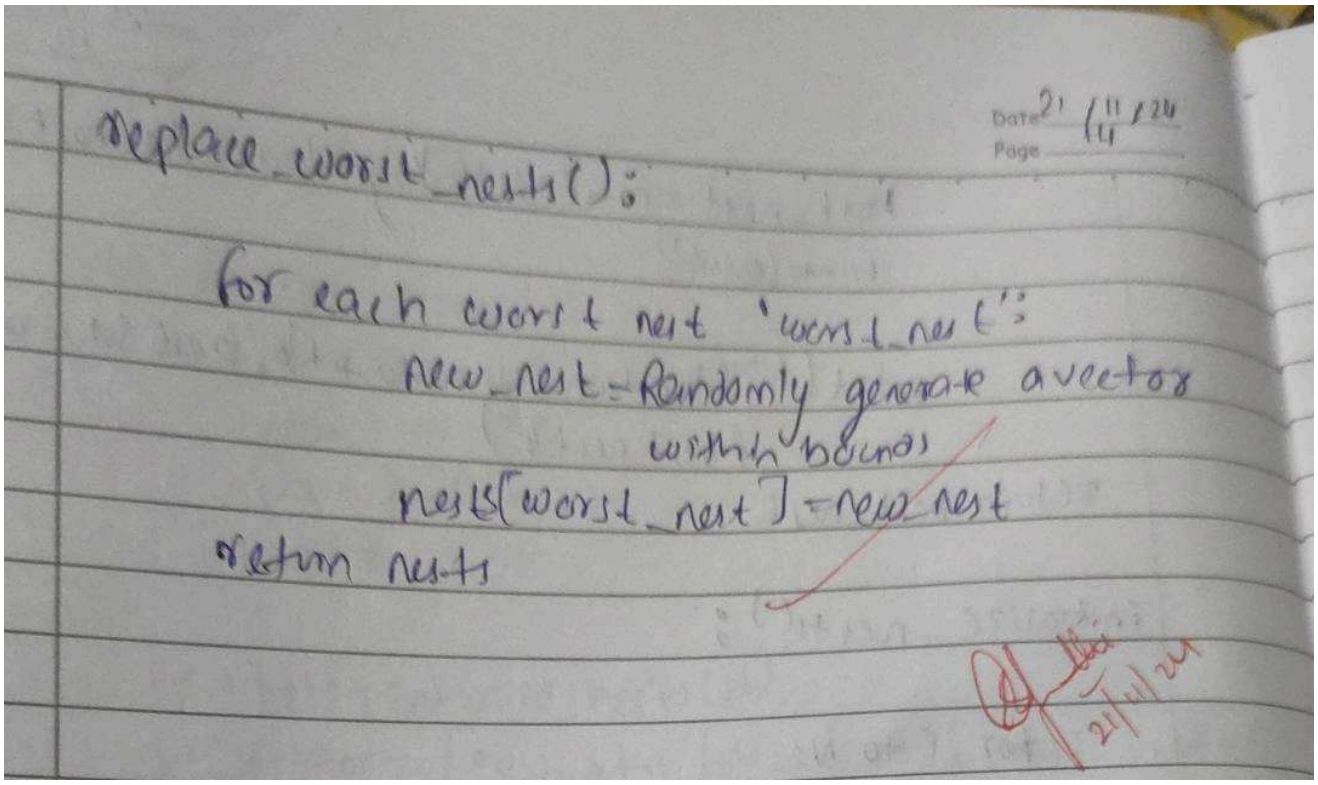
Algorithm:

4) Cuckoo Search

Date 21/11/24
Page 12

```
def cuckoo_search():
    N=50
    max_iter=100
    pa=0.25
    alpha=0.01

    nests=initialize_nests(N)
    fitness=[evaluate_fitness(nest)for nest in nests]

    best_nest=min(nests, key=lambda nest:
    evaluate_fitness(nest)

    for iteration in range(max_iter):
        new_nests=[]
        for nest in nests:
            new_nest=levy_flight(nest,alpha)
            new_nests.append(new_nest)

        new_fitness=[evaluate_fitness(nest) for nest in
            new_nests]

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

        if random.random()<pa:
            worst_nests=select_worst_nests(nest,
                                           fitness
            replace_worst_nests(worst_nests)
```

```
    best_nest = min(nests, key=lambda nest: evaluate_
    fitness(nest))

    print(f"Iteration {iteration+1}, Best solution:
        {best_nest}")
return best_nest

initialize_nests():

    for i to N:
        nest_i = Randomly generate a vector within
                 the bounds: [lowerbound, upper_bound]
        nests[i] = nest_i
    return nests.

evaluate_fitness():
        for a given nest 'nest_i':
        fitness_i = f(nest_i)
        return fitness_i

levy_flight():
        s = Random sample from Gaussian distribution:
                s ~ N(0,1)
        step = alpha * s / |s|^(1/3)
        new_nest_i = nest_i + step
        return new_nest_i

select_worst_nests():
        for given nest and their fitness values:
        sorted_indices = sort(fitness)
        N_worst = floor(N * p_a)
        worst_nests = nests[sorted_indices[N-N_worst
    Return worst_nests.
```



Code:

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

# Objective function: Rastrigin Function
def rastrigin(x):
    A = 10
    return A * len(x) + sum(xi**2 - A * np.cos(2 * np.pi * xi) for xi in x)

# Lévy flight function for generating random steps
def levy_flight(beta=1.5, dim=2):
    sigma_u = np.power(np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) / np.math.gamma((1 +
beta) / 2) / np.power(2, (beta - 1) / 2), 1 / beta)
    sigma_v = 1
    u = np.random.normal(0, sigma_u, dim)
    v = np.random.normal(0, sigma_v, dim)
    return u / np.power(np.abs(v), 1 / beta)

# Cuckoo Search Algorithm
class CuckooSearch:
```

```
    def __init__(self, func, dim, population_size, max_generations, pa=0.25, beta=1.5,
lower_bound=-5, upper_bound=5):
        self.func = func                # Objective function
        self.dim = dim                  # Dimension of the problem
        self.population_size = population_size  # Number of nests (solutions)
        self.max_generations = max_generations  # Maximum number of generations
        self.pa = pa                    # Probability of alien eggs (nest replacement)
        self.beta = beta                # Lévy flight exponent
        self.lower_bound = lower_bound  # Lower bound of the search space
        self.upper_bound = upper_bound  # Upper bound of the search space

        # Initialize population (nests)
        self.nests = np.random.uniform(self.lower_bound, self.upper_bound, (self.population_size,
self.dim))
        self.fitness = np.array([self.func(nest) for nest in self.nests])  # Fitness of each nest
        self.best_nest = self.nests[np.argmin(self.fitness)]  # Best solution found
        self.best_fitness = np.min(self.fitness)  # Best fitness value

    # Update nests using Lévy flights and objective function evaluations
    def generate_new_nests(self):
        new_nests = []
        for i in range(self.population_size):
            step = levy_flight(self.beta, self.dim)
            new_nest = self.nests[i] + step
            # Apply boundary check
            new_nest = np.clip(new_nest, self.lower_bound, self.upper_bound)
            new_nests.append(new_nest)
        return np.array(new_nests)

    # Main cuckoo search algorithm
    def search(self):
        history = []  # To record the best fitness values over generations

        for generation in range(self.max_generations):
            # Generate new nests based on Lévy flight
            new_nests = self.generate_new_nests()
            new_fitness = np.array([self.func(nest) for nest in new_nests])

            # Replace nests with new ones if they are better
            for i in range(self.population_size):
                if new_fitness[i] < self.fitness[i] or np.random.rand() < self.pa:
                    self.nests[i] = new_nests[i]
```

```
                self.fitness[i] = new_fitness[i]

            # Find the best nest in the current population
            current_best_fitness = np.min(self.fitness)
            current_best_nest = self.nests[np.argmin(self.fitness)]

            # Update the global best solution
            if current_best_fitness < self.best_fitness:
                self.best_fitness = current_best_fitness
                self.best_nest = current_best_nest

            # Record the best fitness for the current generation
            history.append(self.best_fitness)
            print(f"Generation {generation+1}: Best fitness = {self.best_fitness}")

        return self.best_nest, self.best_fitness, history

# Analyze the Cuckoo Search Algorithm
def analyze_cuckoo_search():
    # Set up parameters for Cuckoo Search
    dim = 2
    population_size = 50
    max_generations = 100
    cuckoo_search = CuckooSearch(func=rastrigin, dim=dim, population_size=population_size,
max_generations=max_generations)

    # Run the Cuckoo Search algorithm
    best_nest, best_fitness, history = cuckoo_search.search()

    # Plot the convergence curve
    plt.plot(history)
    plt.title("Convergence Curve of Cuckoo Search Algorithm")
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness")
    plt.show()

    print(f"Best solution found: {best_nest}")
    print(f"Best fitness: {best_fitness}")

# Run the analysis
analyze_cuckoo_search()
```

Output:

```
Best solution found: [1.30548027 2.02026344]
Best fitness: 0.16306139523513963
```

Program 5

Grey Wolf Optimizer

Algorithm:

Date 28.11.24
Page 16

5) Grey Wolf Optimizer (GWO) :

Initialize Parameters:
- Set N (number of wolves), D (dimensions), Max_Itr (max iterations), bounds.

Initialize Population:
- Randomly initialize the positions of N wolves in D-dimensional space.

Evaluate fitness
- for each wolf, calculate the fitness using the objective function.

Identify Alpha, Beta, Delta wolves:
- sort wolves by fitness.
- Alpha ← best wolf
- Beta ← second best wolf
- Delta ← third best wolf

Main optimization Loop (for t = 1 to Max_Itr):
- Calculate the decreasing factor a = 2 - t*(2/Max_Itr)
- for each wolf i:
  - Calculate the fitness of wolf i
  - If fitness of wolf i < fitness of alpha
    - Delta ← Beta
    - Beta ← Alpha
    - Alpha ← wolf i (update alpha position and fitness)
- Update position of Each wolf:
  for each wolf i:
  - Calculate A = 2*r1 - a
  - Calculate C = 2*r2

Date 23/4/24
Page 16

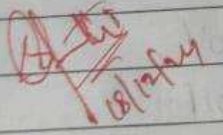
- Calculate D = |C * Alpha_position of wolf[i]
- Update position of wolf i: position[i] = Alpha - A * D
- Ensure position[i] stays within bounds

Check convergence (optional):
- If stopping criteria are met, stop early.

Output Best solution:
- The position of the Alpha wolf i.e the Best solution find

Objective function:
return np.sum(x**2)

5(b)



Code:

```
import numpy as np

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

class GreyWolfOptimizer:
    def __init__(self, objective_function, n_wolves, n_variables, max_iter, lb, ub):
        self.obj_func = objective_function  # Objective function
        self.n_wolves = n_wolves  # Number of wolves
        self.n_variables = n_variables  # Number of variables in the problem
        self.max_iter = max_iter  # Maximum number of iterations
        self.lb = lb  # Lower bound for the search space
        self.ub = ub  # Upper bound for the search space

        self.wolves = np.random.uniform(self.lb, self.ub, (self.n_wolves, self.n_variables))

        self.alpha = np.zeros(self.n_variables)
        self.beta = np.zeros(self.n_variables)
        self.delta = np.zeros(self.n_variables)
        self.alpha_score = float("inf")
        self.beta_score = float("inf")
        self.delta_score = float("inf")

    def update_wolves(self):
        fitness = np.apply_along_axis(self.obj_func, 1, self.wolves)

        sorted_indices = np.argsort(fitness)
        self.wolves = self.wolves[sorted_indices]
        fitness = fitness[sorted_indices]

        # Update alpha, beta, and delta wolves
        self.alpha = self.wolves[0]
        self.beta = self.wolves[1]
        self.delta = self.wolves[2]
        self.alpha_score = fitness[0]
        self.beta_score = fitness[1]
        self.delta_score = fitness[2]

    def optimize(self):
```

```
        for t in range(self.max_iter):
            A = 2 * np.random.random((self.n_wolves, self.n_variables)) - 1  # Random values for exploration
            C = 2 * np.random.random((self.n_wolves, self.n_variables))  # Random values for exploitation
            for i in range(self.n_wolves):
                D_alpha = np.abs(C[i] * self.alpha - self.wolves[i])  # Distance to alpha wolf
                D_beta = np.abs(C[i] * self.beta - self.wolves[i])  # Distance to beta wolf
                D_delta = np.abs(C[i] * self.delta - self.wolves[i])  # Distance to delta wolf

                self.wolves[i] = self.alpha - A[i] * D_alpha

                self.wolves[i] = np.clip(self.wolves[i], self.lb, self.ub)

            self.update_wolves()

            print(f"Iteration {t+1}/{self.max_iter}, Best Score: {self.alpha_score}")

        return self.alpha, self.alpha_score  # Return the best solution found

n_wolves = 30  # Number of wolves
n_variables = 5  # Number of decision variables
max_iter = 100  # Maximum number of iterations
lb = -10  # Lower bound of the search space
ub = 10  # Upper bound of the search space

gwo = GreyWolfOptimizer(objective_function, n_wolves, n_variables, max_iter, lb, ub)
best_solution, best_score = gwo.optimize()
print("Best Solution Found:", best_solution)
print("Best Score:", best_score)
```

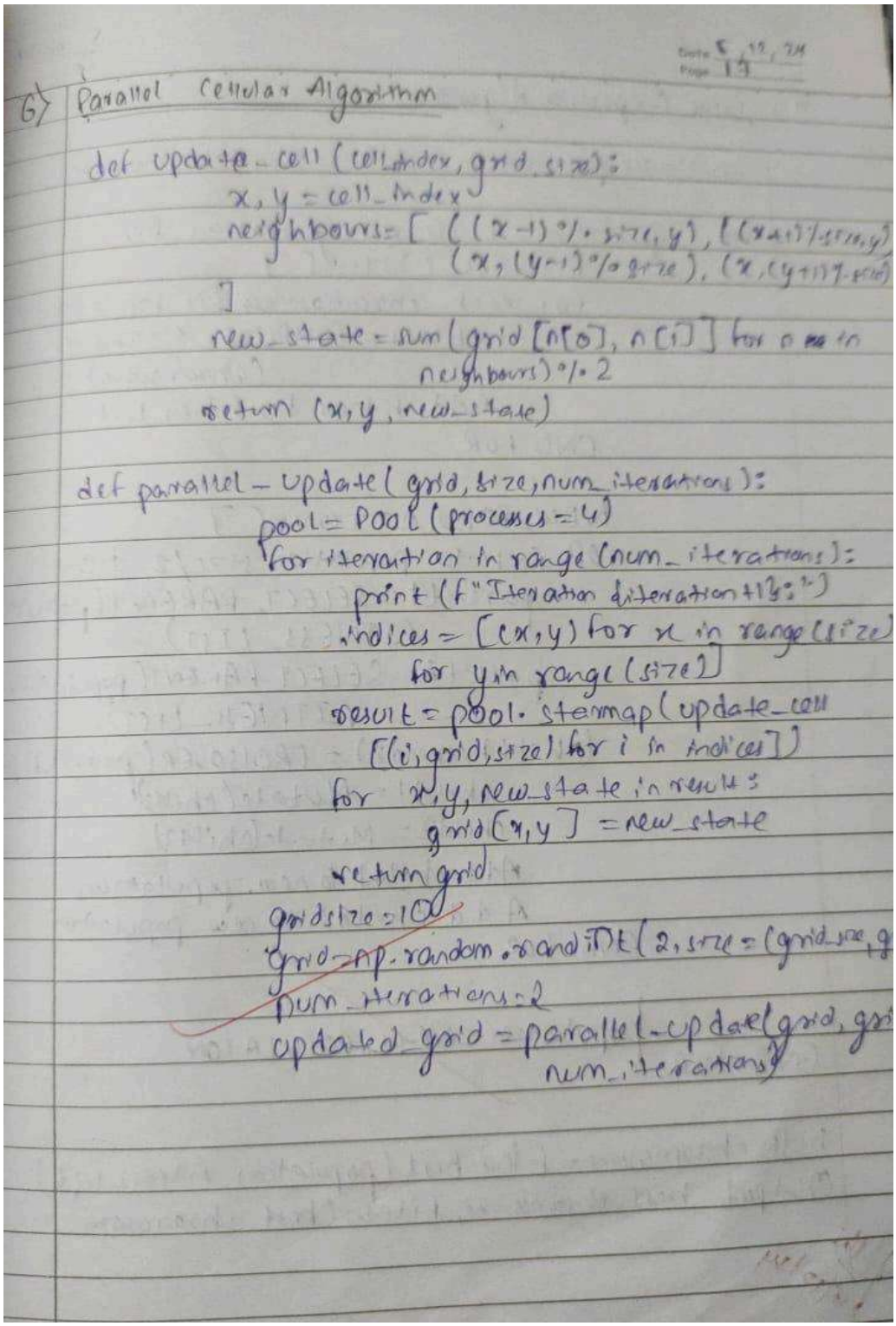
Output:

```
Iteration 100/100, Best Score: 1.985808550535119e-30
Best Solution Found: [-4.38373504e-17 -4.54363691e-16 -1.31663573e-15 -2.05502414e-16
  4.09828696e-17]
Best Score: 1.985808550535119e-30
```

Program 6

Parallel Cellular Algorithm

Algorithm:



Code:

```
import numpy as np
from multiprocessing import Pool
def update_cell(cell_index, grid, size):
```

```
    x, y = cell_index
    neighbors = [
        ((x-1) % size, y), ((x+1) % size, y),
        (x, (y-1) % size), (x, (y+1) % size)
    ]
    new_state = sum(grid[n[0], n[1]] for n in neighbors) % 2  # example: majority rule
    return (x, y, new_state)
def parallel_update(grid, size, num_iterations):
    pool = Pool(processes=4)
    for iteration in range(num_iterations):
        print(f"Iteration {iteration + 1}:")
        indices = [(x, y) for x in range(size) for y in range(size)]
        result = pool.starmap(update_cell, [(i, grid, size) for i in indices])

        for x, y, new_state in result:
            grid[x, y] = new_state
        print(grid)
    return grid
grid_size = 10
grid = np.random.randint(2, size=(grid_size, grid_size))
print("Initial state:")
print(grid)
num_iterations = 2
updated_grid = parallel_update(grid, grid_size, num_iterations)
```

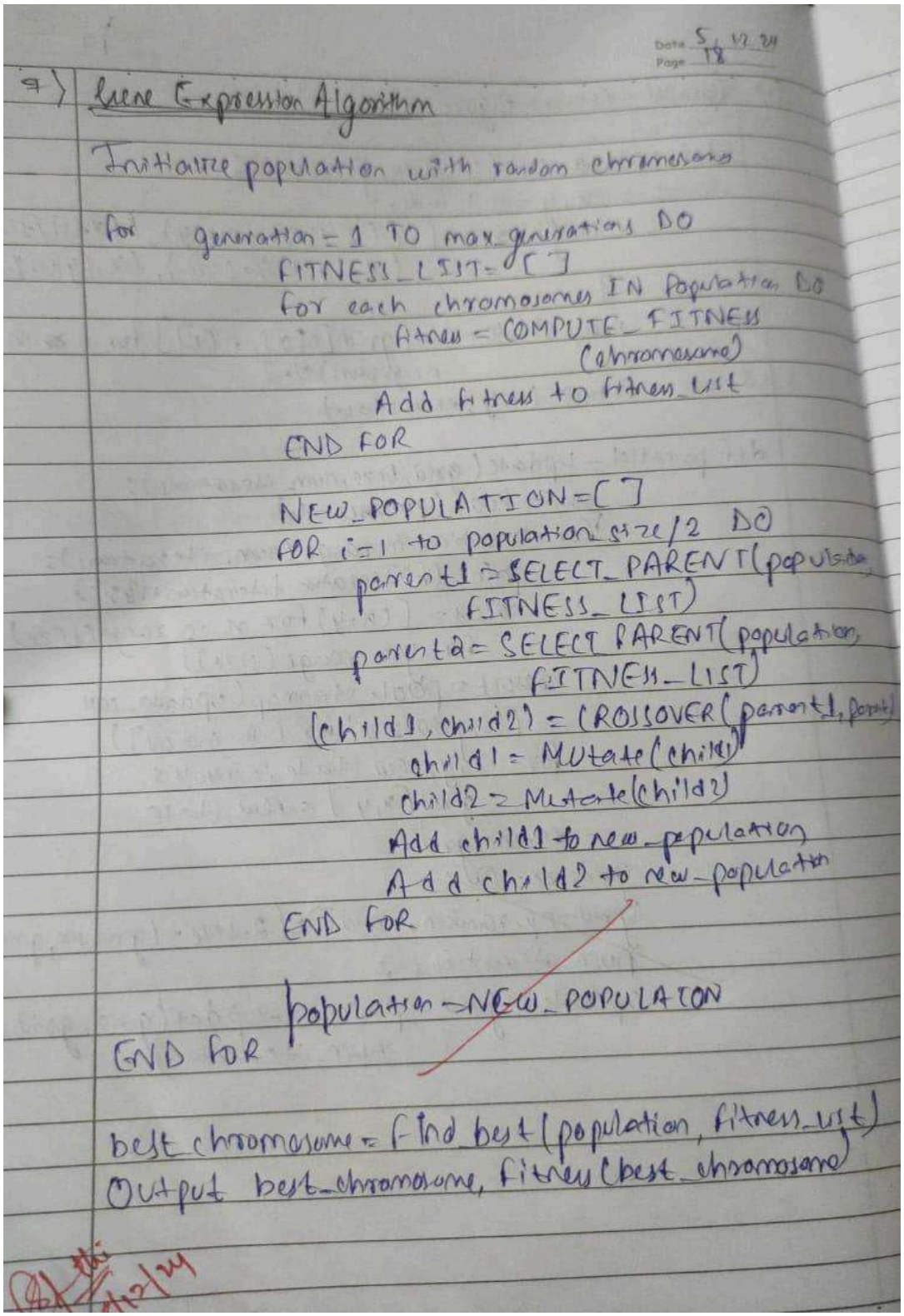
Output:

```
Iteration 1:
[[1 0 0 1]
 [1 0 1 0]
 [1 0 0 1]
 [0 1 0 1]]
Iteration 2:
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

Program 7

Gene Expression Algorithm

Algorithm:



7) Gene Expression Algorithm

Initialize population with random chromosome

for generation = 1 TO max_generations DO
    FITNESS_LIST = [ ]
    for each chromosome IN Population DO
        fitness = COMPUTE_FITNESS (chromosome)
        Add fitness to fitness_list
END FOR

NEW_POPULATION = [ ]
FOR i = 1 to population_size/2 DO
    parent1 = SELECT_PARENT(population, FITNESS_LIST)
    parent2 = SELECT PARENT(population, FITNESS_LIST)
    (child1, child2) = CROSSOVER(parent1, parent2)
    child1 = Mutate(child1)
    child2 = Mutate(child2)
    Add child1 to new_population
    Add child2 to new_population
END FOR

population = NEW_POPULATION

END FOR

best_chromosome = find_best(population, fitness_list)
Output best_chromosome, fitness(best_chromosome)

Code:

```
import random
import numpy as np
import operator

# Function set and terminal set
FUNCTIONS = {'+': operator.add, '-': operator.sub, '*': operator.mul, '/': operator.truediv}
TERMINALS = ['x', 1, 2, 3, 4]  # x and constants

def random_gene(length=10):
    """Generate a random chromosome (gene)."""
    return [random.choice(list(FUNCTIONS.keys()) + TERMINALS) for _ in range(length)]


def decode_chromosome(chromosome, x):
    """Decode chromosome into a functional expression tree (phenotype)."""
    stack = []
    for gene in chromosome:
        if gene in FUNCTIONS:  # If it's a function, pop arguments and apply
            if len(stack) < 2:  # Avoid errors if stack has fewer than 2 elements
                stack.append(0)
                continue
            b = stack.pop()
            a = stack.pop()
            try:
                result = FUNCTIONS[gene](a, b)
            except ZeroDivisionError:
                result = 1  # Avoid division by zero
            stack.append(result)
        elif gene == 'x':
            stack.append(x)
        else:
            stack.append(gene)
    return stack[0] if stack else 0  # Return top of stack as output


def fitness_function(chromosome, target_function, x_values):
    """Calculate fitness based on Mean Squared Error."""
    predictions = [decode_chromosome(chromosome, x) for x in x_values]
    targets = [target_function(x) for x in x_values]
    mse = np.mean([(p - t) ** 2 for p, t in zip(predictions, targets)])
    return mse
```

```
def selection(population, fitnesses):
    """Select individuals based on fitness (roulette wheel selection)."""
    total_fitness = sum(1 / (f + 1e-6) for f in fitnesses)  # Avoid division by zero
    probabilities = [(1 / (f + 1e-6)) / total_fitness for f in fitnesses]
    return population[np.random.choice(len(population), p=probabilities)]


def mutate(chromosome, mutation_rate=0.1):
    """Apply mutation to a chromosome."""
    new_chromosome = chromosome[:]
    for i in range(len(new_chromosome)):
        if random.random() < mutation_rate:
            new_chromosome[i] = random.choice(list(FUNCTIONS.keys()) + TERMINALS)
    return new_chromosome


def crossover(parent1, parent2):
    """Perform one-point crossover between two parents."""
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2


def gene_expression_algorithm(target_function, x_values, population_size=10, generations=20):
    """Main Gene Expression Algorithm."""
    # Initialize random population
    population = [random_gene() for _ in range(population_size)]

    print("Initial Population:")
    for i, chrom in enumerate(population):
        print(f"Chromosome {i}: {chrom}")

    for generation in range(generations):
        print(f"\nGeneration {generation + 1}:")
        # Calculate fitness for each individual
        fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
        for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
            print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

        # Select the next generation
```

```
        new_population = []
        for _ in range(population_size // 2):
            parent1 = selection(population, fitnesses)
            parent2 = selection(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population

    # Final results
    print("\nFinal Population and Fitness:")
    fitnesses = [fitness_function(chrom, target_function, x_values) for chrom in population]
    for i, (chrom, fit) in enumerate(zip(population, fitnesses)):
        print(f"Chromosome {i}: {chrom}, Fitness: {fit:.4f}")

    best_index = np.argmin(fitnesses)
    print("\nBest Solution:")
    print(f"Chromosome: {population[best_index]}, Fitness: {fitnesses[best_index]:.4f}")

# Target function for regression
def target_function(x):
    return x**2 + 2*x + 1  # Example: f(x) = x^2 + 2x + 1

# Input values
x_values = np.linspace(-10, 10, 20)

# Run the algorithm
gene_expression_algorithm(target_function, x_values, population_size=10, generations=10)
```

Output:

```
Best Solution:
Chromosome: [1, 3, '+', 2, 1, 4, '*', '*', '*', 3], Fitness: 1259.2067
<ipython-input-3-6df17022c257>:25: RuntimeWarning: divide by zero encountered in scalar divide
  result = FUNCTIONS[gene](a, b)
```