VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Shreyansh Sethiya (1BM22CS269)

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 **Shreyansh Sethiya (1BM22CS269)** 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. Spoorthi D M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/Shreyanshsethiya/BISLAB

Program 1

Genetic Algorithm for Optimization Problems

Algorithm:

29/10/24

Genetic Algorithm

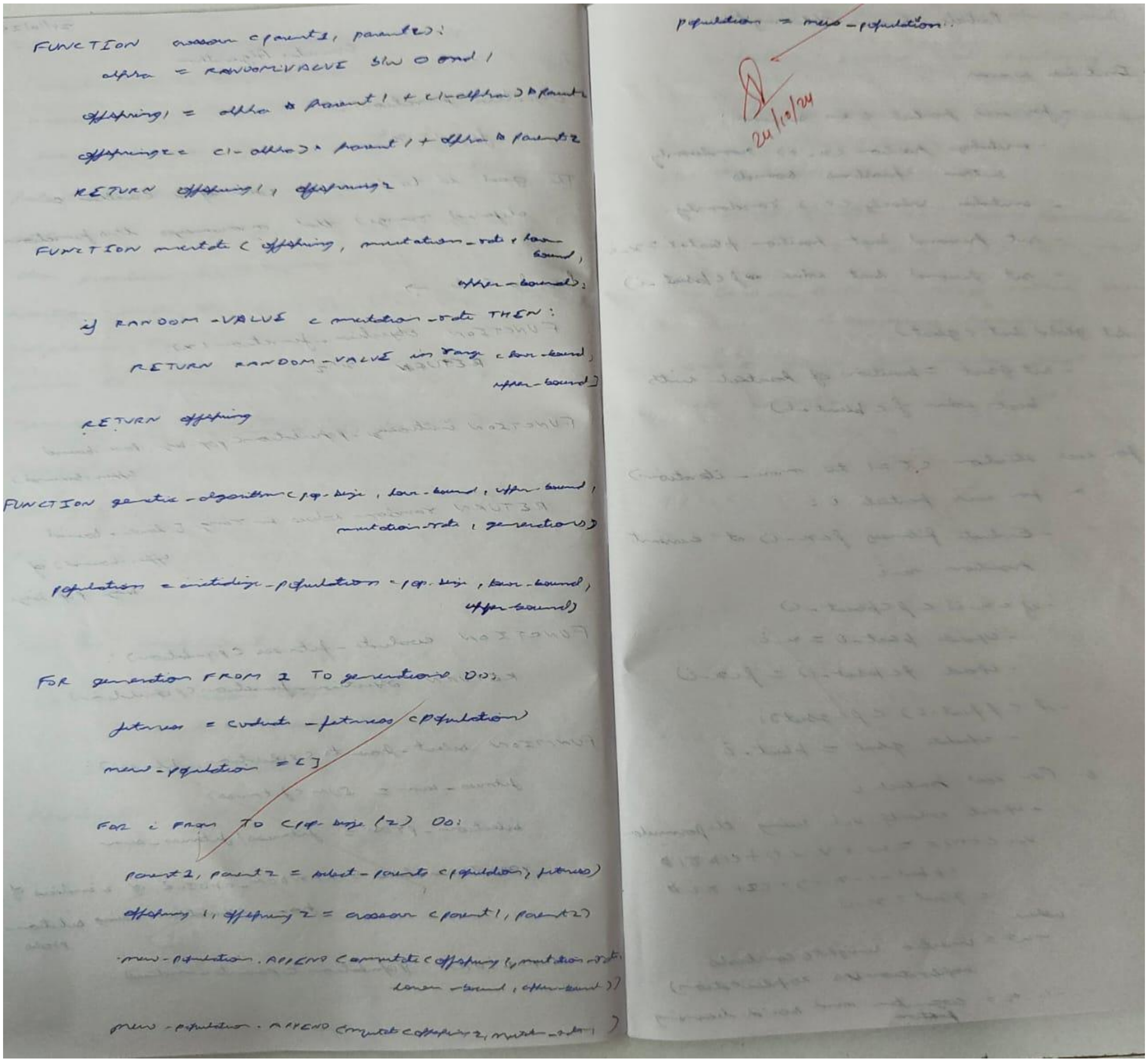
Objective function:

$f(x) = x^2$

The goal is to find the value of x (within a defined range) that maximizes this function

Pseudo code →

```
FUNCTION objective_function (x):
    RETURN x^2

FUNCTION initialize_population(pop_size, lower_bound,
                                upper_bound):

    RETURN random values in range [lower_bound,
                                   upper_bound] of
                                   size pop_size

FUNCTION evaluate_fitness (population):
    RETURN objective_function(population)

FUNCTION select_parents (population, fitness):
    fitness_sum = SUM (fitness)
    selection_probs = fitness / fitness_sum
    parents_indices = RANDOM_CHOICE of 2 indices of
                      from population using selection_
                      probs

    RETURN population (parents_indices)
```


FUNCTION crossover(parent1, parent2):
alpha = RANDOM_VALUE b/w 0 and 1
offspring1 = alpha * parent1 + (1-alpha) * parent2
offspring2 = (1-alpha) * parent1 + alpha * parent2
RETURN offspring1, offspring2

FUNCTION mutate(offspring, mutation_rate, lower_bound, upper_bound):
if RANDOM_VALUE < mutation_rate THEN:
RETURN RANDOM_VALUE in range (lower_bound, upper_bound)
RETURN offspring

FUNCTION genetic_algorithm(pop_size, lower_bound, upper_bound, mutation_rate, generations)
population = initialize_population(pop_size, lower_bound, upper_bound)
FOR generation FROM 1 TO generations DO:
fitness = evaluate_fitness(population)
new_population = []
FOR i FROM 1 TO (pop_size / 2) DO:
parent1, parent2 = select_parents(population, fitness)
offspring1, offspring2 = crossover(parent1, parent2)
new_population.APPEND(mutate(offspring1, mutation_rate, lower_bound, upper_bound))
new_population.APPEND(mutate(offspring2, mutation_rate,)
population = new_population
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:

Ant Colony Optimization for Travelling Salesman

Objective → To find the shortest possible route of salesman that visits a list of cities and returns to the origin city using ant colony

1) Initialization →

- Number of Ants → A fixed number of ants are used to explore the solution space. Each ant is treated as an individual agent to construct a solution.
- Number of iterations → Runs for a fixed number of iterations.

Pheromone Parameters:

- alpha : ~~weight~~ concentration of the pheromone in the decision-making process
- beta : weight of the heuristic (inverse distance) in the decision-making process.
- rho : Evaporation rate of the pheromone, which simulates how old pheromones dissipate over time.

Distance Matrix : Euclidean distance btw each pair of cities is pre-computed

2) Construction of Ant Tours :→

- For each ant a tour is constructed:

Each ant starts at a random city

From the starting city, it repeatedly choses the next city to visit based on the following factors →

Pheromone concentration → The stronger the pheromone on an edge

Heuristic information → The inverse of the distance b/w cities. The shorter the distance, the more likely an ant to choose that edge.

Probability calculation → (Alpha) α and (heuristic information) β to calculate the probability

Ant select the ants with greater probability. and a degree of randomness is also include to encourage exploration.

This process continues until all cities are visited.

write pseudocode

24/10/24

3) Pheromone Update:

- After all ants have completed their tours, the pheromone matrix is updated →

Pheromone Evaporation: All pheromone values are reduced by a factor (1 - rho), simulating the natural evaporation of pheromone over time

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

1) Initialize swarm

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

2) Set global best (gbest)

- set gbest = position of particle with best value f(pbest_i)

3) for each iteration (t=1 to max_iteration)

a. for each particle i:

- Evaluate fitness f(x_i) at current position x_i
- if (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

b. For each particle i

- update velocity v_i using the formula

$v_i(t+1) = w + v_i(t) + c_1 * r_1 *$
$(pbest_i - x_i) + c_2 * r_2 *$
$(gbest - x_i)$

where:

- w = inertia weight (controls exploration vs exploitation)
- c_1, c_2 = cognitive and social learning factors

- r1, r2 = random value b/w 0 and 1

- pbest_i = personal best position of particle i

- gbest = global best position of the swarm

c. For each particle i

- update position x_i using formula

$$x\_i(t+1) = x\_i(t) + v\_i(t+1)$$

- Ensure that x_i stays within bounds (boundary handling)

4) Output global best position gbest and its corresponding value f(gbest)

07/11/24

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:

Cuckoo Search Algorithm

Pseudo code ⇒

```
#1) Define the objective function to be optimized
    def objective_function(x):

#2) Initialize Parameters
    N = 20            # Number of Nests
    pa = 0.25         # Probability of discovery
    Max Iter = 100    # Maximum number of iterations
    dimensions = 2    # Dimensionality of the problem
    Lower Bound = -5  # Lower bound of search space
    Upper Bound = 5   # Upper bound of search space

#3) Initialize population (nests)
    def initialize_population(N, dimensions, lower bound,
                                              upper bound):
        # random position within the bounds

#4) Evaluate Fitness (Calculate for each nest :)
    def evaluate_fitness(nests):
            return np.apply_along_axis(objective_function, 1,
                                                      nests)

#5) Generate New solutions using Levy flights ⇒
    def levy_flight(dimension, beta=1.5):
        # Returns random step size using levy distribution
        step = u/np.abs(v) ** (1/beta)
        return step
```

Generate new positions using levy flight for each nest.

```
def generate_new_solution(nest, alpha = 0.01):
    # Apply levy flight to generate new position
    return nest + alpha * levy_flight(dimension)
```

#6 Bound the solution within the allowed limit

```
def bound_solution(solution, LowerBound, UpperBound):
    return np.clip(solution, LowerBound, UpperBound)
```

#7 Main loop for iterations →

```
for iter in range(MaxIter):
    for i in range(N):
        # Generate new solution for nest i
        new_nest = generate_new_solution(nests[i])

        # Bound the new solution
        new_nest = bound_solution(new_nest,
                                  LowerBound, UpperBound)

        # Evaluate the fitness of the new solution
        new_fitness = objective_function
                                  (new_nest)

        # If the new solution is better, replace
            the old one
        if new_fitness < fitness[i]:
            nests[i] = new_nest
            fitness[i] = new_fitness
```

21/11/24

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:

Gray Wolf Optimizer

Pseudo Code →

// Step 1: Define the Problem

objective_Function(x) // Define the objective function
// to be optimized

// Step 2: Initialize Parameters

N = Number of wolves

max_iter = Maximum number of iterations.

Dim = Number of dimensions (variables of the problems)

lower_bound = lower bound of the search space

upper_bound = Upper bound of the search space

// Step 3: Initialize Problem

for i=1 to N:

Wolf [i].Position = Random (lower_bound, upper_bound)
// Initialize position of wolves

Wolf [i].Fitness = objective_Function (Wolf [i].Position)
// Evaluate fitness of each wolf

// Step 4: Sort the wolves by fitness

Sort wolves based on fitness in ascending order

Alpha wolf = Wolf [1] // Best fitness
Beta wolf = Wolf [2] // Second-best fitness
Delta wolf = Wolf [3] // Third-best fitness

// Step 5: Main optimization loop

for iteration = 1 to max_iter:

for i = 1 to N:

// Step 5.1: Update Position of each wolf

a = 2 * Random (0, 1) - 1 // Random vector for exploration

c = 2 * Random (0, 1)

// Calculate Distance b/w current wolf and alpha, beta, delta wolves →

D_alpha = abs (C * Alpha_Wolf.Position - Wolf[i].Position)

D_beta = abs (C * Beta_Wolf.Position - Wolf[i].Position)

D_delta = abs (C * Delta_Wolf.Position - Wolf[i].Position

// Update Position of current wolf

Wolf [i].Position = Wolf [i].Position + a * (D_alpha + D_beta + D_delta)

// Ensure the new position is within bound

for j = 1 to Dim:

If Wolf[i].Position[j] < Lower_bound [j];

Wolf [i].Position [j] = Lower_bound [j]

Else if Wolf [i].Position [j] > Upper_bound [j];

Wolf[i].Position [j] = Upper_bound [j]

// Step 5.2 Evaluate fitness of the updated position

Wolf [i].Fitness = objective_function (Wolf [i].Position)

// Step 5.3 : Sort the wolves again based on fitness

Sort wolves based on fitness in ascending order

AlphaWolf = Wolf [1]

BetaWolf = Wolf [2]

Delta Wolf = Wolf [3]

// Step 6 : Output the Best solution

Bestsolution = AlphaWolf. Position

Best fitness = AlphaWolf. fitness

Return Bestsolution, Best fitness.

o/p?

18/12/24

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
```

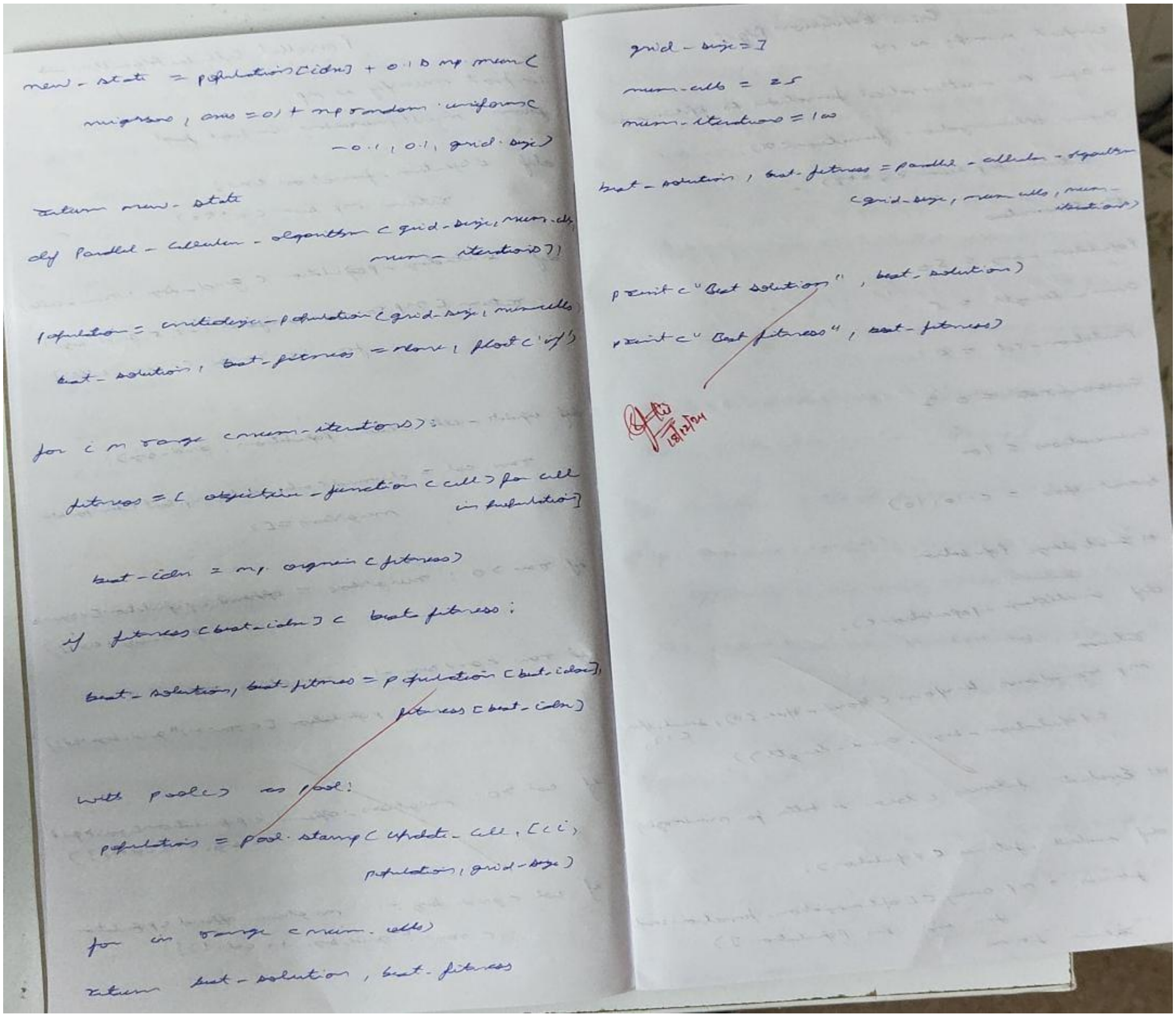
Program 6

Parallel Cellular Algorithm

Algorithm:

Parallel Cellular Algorithms →

```
import numpy as np
from multiprocessing import pool

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

def initialize_population (grid_size, num_cells):
    return [np.random.uniform (-5, 5, grid_size)
                for in range (num_cells)]

def update_cell (idx, population, grid_size):
    row, col = divmod (idx, int(np.sqrt (len(population
                neighbors = []

if row > 0 : neighbors = append (population[(row-1)*
                        grid_size + col])

if row < grid_size - 1 : neighbors . append (
            population [(row+1)* grid_size + col])

if col > 0 : neighbors . append (population [row*grid
                    _size + (col-1)])

if col < grid_size - 1 : neighbors . append (population
        (row * grid_size + (col+1)])
```



new_state = population[idx] + 0.1 * np.mean(
    neighbors, axis=0) + np.random.uniform(
                    -0.1, 0.1, grid_size)

return new_state

def parallel_cellular_algorithm(grid_size, num_cells,
                    num_iterations):

population = initialize_population(grid_size, num_cells)
best_solution, best_fitness = None, float('inf')

for i in range(num_iterations):

fitness = [objective_function(cell) for cell
                    in population]

best_idx = np.argmin(fitness)

if fitness[best_idx] < best_fitness:

best_solution, best_fitness = population[best_idx],
                    fitness[best_idx]

with Pool() as pool:

population = pool.starmap(update_cell, [(i,
                    population, grid_size)

for in range(num_cells)

return best_solution, best_fitness

grid_size = 7

num_cells = 25

num_iterations = 100

best_solution, best_fitness = parallel_cellular_algorithm
                    (grid_size, num_cells, num_
                    iterations)

print("Best solution:", best_solution)

print("Best fitness:", best_fitness)

18/12/24

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]]
```

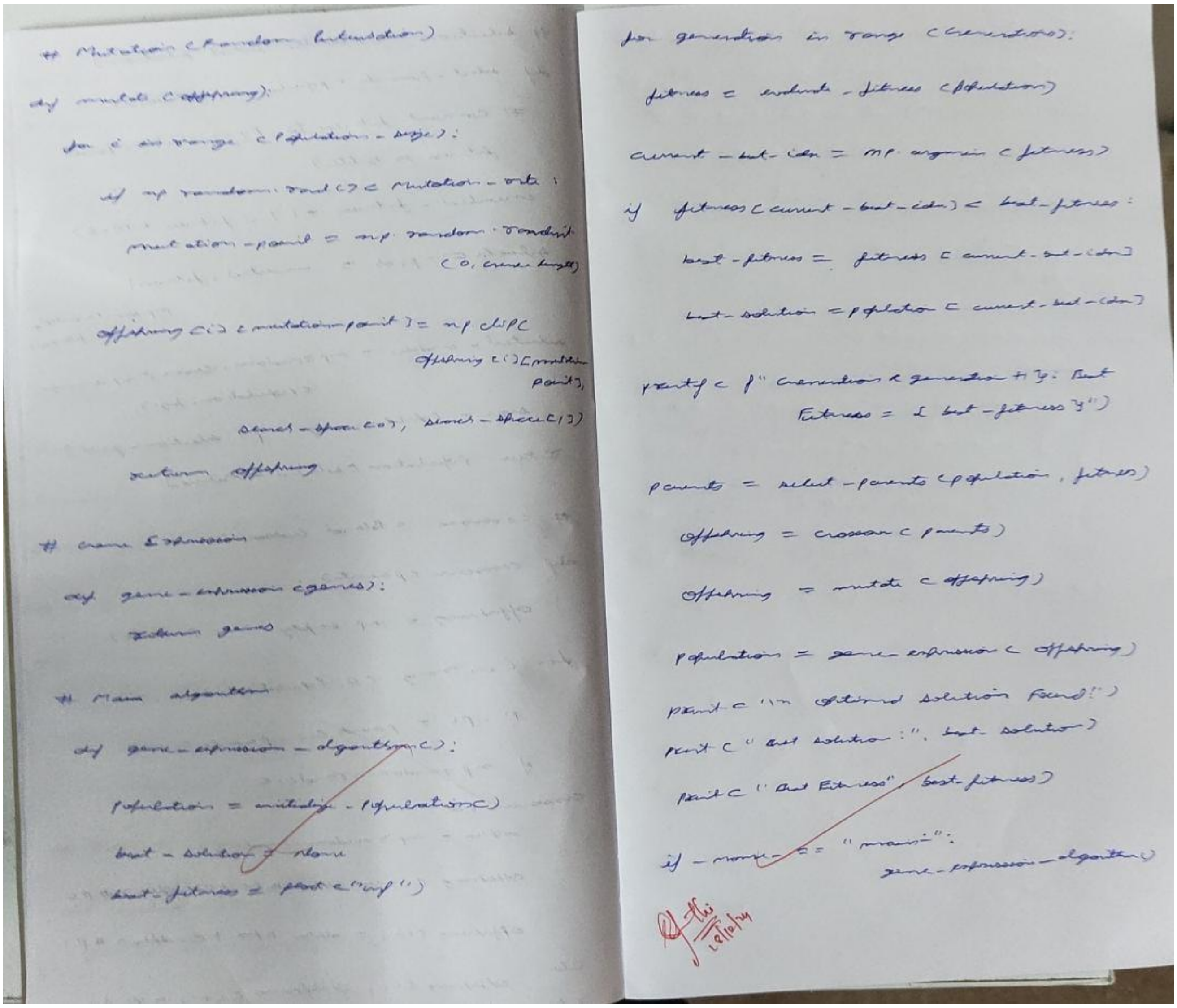
Program 7

Gene Expression Algorithm

Algorithm:

Gene Expression Algo:

```
import numpy as np
# define a mathematical function to optimize
def optimization_function(x):
    return np.sum(x**2)

# Parameters
population_size = 50
gene_length = 5
mutation_rate = 0.1
crossover_rate = 0.7
generations = 100
search_space = (-10, 10)

# Initialize Population
def initialize_population():
    return
    np.random.uniform(search_space[0], search_space[1],
        (population_size, gene_length))

# Evaluate fitness (lower is better for minimizing)
def evaluate_fitness(population):
    fitness = np.array([optimization_function(ind)
        for ind in population])
    return fitness

# Selection (Roulette wheel selection)
def select_parents(population, fitness):
    # convert fitness to probabilities (lower
      fitness is better)
    inverted_fitness = 1 / (fitness + 1e-6)
    selection_probs = inverted_fitness /
                      np.sum(inverted_fitness)

    selected_indices = np.random.choice(np.arange
                       (population_size),
        size = population_size, p = selection_probs)
    return population[selected_indices]

# Crossover (Blend Crossover)
def crossover(parents):
    offspring = np.empty_like(parents)
    for i in range(0, population_size, 2):
        p1, p2 = parents[i], parents[i+1]
        if np.random.rand() <
crossover_rate:
            alpha = np.random.rand()
            offspring[i] = alpha * p1 + (1-alpha) * p2
            offspring[i+1] = alpha * p2 + (1-alpha) * p1
        else offspring[i], offspring[i+1] = p1, p2
    return offspring
```



```
# Mutation (Random Substitution)
def mutate (offspring):
    for i in range (Population_size):
        if np.random.rand() < Mutation_rate:
            mutation_point = np.random.randint
                                    (0, Gene_length)
            offspring[i][mutation_point] = np.clip(
                                    offspring[i][mutation
                                    point],
                        Search_space[0], Search_space[1])
        return offspring

# Gene Expression
def gene_expression (genes):
    return genes

# Main algorithm
def gene_expression_algorithm():
    population = initialize_Population()
    best_solution = None
    best_fitness = float("inf")
for generation in range (Generations):
    fitness = evaluate_fitness (population)
    current_best_idx = np.argmin (fitness)
    if fitness[current_best_idx] < best_fitness:
        best_fitness = fitness[current_best_idx]
        best_solution = population[current_best_idx]

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

    parents = select_parents (population, fitness)
    offspring = crossover (parents)
    offspring = mutate (offspring)

    population = gene_expression (offspring)
    print ("\n Optimized Solution Found!")
    print ("Best Solution:", best_solution)
    print ("Best Fitness", best_fitness)

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

28/10/24

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)]
```

```python
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)
```