VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Bio Inspired Systems (23CS5BSBIS)

Submitted by

Samraat Dabolay (1BM22CS236)

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 **Samraat Dabolay (1BM22CS236),** who is a 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/samraatd/BIS-LAB

Program 1: Genetic Algorithm for Optimization Problems

Problem Statement: Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

Algorithm:

24/10/24

Algorithm 1 : Genetic Algorithm for Optimisation Problems

```
Initialise_population ( bounds , n ) {
        random.uniform ( bounds [0], bounds [1] , 1 ) }

Evaluate - fitness ( population ) {
        ~~objective~~ function ( i ) for i in population }

Roulette - wheel ( pop , scores ) {
        total ← sum (scores)
        prob ← 1 - score / total for each score
        selection ← random.choice ( len (pop) , p )
        p ← prob / sum (prob)
        return pop [ selection ]

Crossover ( p1 , p2 , α ):
    return offspring = alpha * p1 [0] + (1 - α) * p2[0

Mutation ( individuals , bounds , rate ) :
    if random ( ) < rate :
        return [ random.uniform ( bounds [0], bounds[
    return individual

Genetic - Algorithm ( bounds , iter , n-pop , rate ) :
    pop ← initialise population ( bounds , n - pop )
    best , best - eval ← pop [0] , function ( pop [0] )
```

for gen in range (iter):
scores ← evaluate - fitness (pop)
for i in range (n - pop):
if scores[i] < best - eval:
best, best - eval ← pop[i], scores[i]

children = [ ]
for _ in range (n - pop):
p1 ← roulette - wheel (pop, scores)
p2 ← roulette - wheel (pop, scores)

child ← crossover (p1, p2)
child ← mutation (child, bounds, rate)
Add child to children list
pop ← children
return [best, best - eval]

Input:
Range / bounds = [-10, 10]
Total iterations = 50
Population size = 100
Mutation rate = 0.1
Alpha = 0.5

24/10/24

Code:

```
import numpy as np


# Objective function: f(x) = x^2
def objective(x):
    return x[0] ** 2 + 2*x[0] + 1
```

```
# Initialization: generate initial population
def initialize_population(bounds, n_pop):
    return [np.random.uniform(bounds[0], bounds[1], 1).tolist() for _ in range(n_pop)]


# Fitness evaluation
def evaluate_fitness(pop):
    return [objective(ind) for ind in pop]


# Roulette wheel selection
def roulette_wheel_selection(pop, scores):
    total_fitness = sum(scores)
    probabilities = [1 - (score / total_fitness) for score in scores]
    selection_ix = np.random.choice(len(pop), p=np.array(probabilities) / sum(probabilities))
    return pop[selection_ix]


# Crossover: linear combination of parents
def crossover(p1, p2, alpha=0.5):
    offspring = alpha * p1[0] + (1 - alpha) * p2[0]
    return [offspring]  # Ensure offspring is a list


# Mutation: random value within bounds
def mutation(individual, bounds, r_mut):
    if np.random.rand() < r_mut:
        return [np.random.uniform(bounds[0], bounds[1])]
    return individual


# Genetic algorithm
def genetic_algorithm(bounds, n_iter, n_pop, r_mut):
    # Initialize population
    pop = initialize_population(bounds, n_pop)
    best, best_eval = pop[0], objective(pop[0])


    for gen in range(n_iter):
        # Evaluate fitness
        scores = evaluate_fitness(pop)
```

```
        # Check for new best solution
        for i in range(n_pop):
            if scores[i] < best_eval:
                best, best_eval = pop[i], scores[i]
                print(f">{gen}, new best f({pop[i]}) = {scores[i]:.6f}")


        # Select parents and create offspring
        children = []
        for _ in range(n_pop):
            p1 = roulette_wheel_selection(pop, scores)
            p2 = roulette_wheel_selection(pop, scores)
            offspring = crossover(p1, p2)
            offspring = mutation(offspring, bounds, r_mut)  # Pass as list
            children.append(offspring)


        # Replace population with new offspring
        pop = children


    return [best, best_eval]


# Define range for input
bounds = [-10.0, 10.0]
# Define the total iterations
n_iter = 50
# Define the population size
n_pop = 100
# Mutation rate
r_mut = 0.1


# Perform the genetic algorithm search
best, score = genetic_algorithm(bounds, n_iter, n_pop, r_mut)
print('Done!')
print(f'f({best}) = {score:.6f}')
```

Output:

```
>0, new best f([-5.666534385599229]) = 21.776543
>0, new best f([1.080175298094792]) = 4.327129
>0, new best f([-1.5931328574092856]) = 0.351807
>0, new best f([-1.5653111777364508]) = 0.319577
>0, new best f([-0.8482241989814483]) = 0.023036
>0, new best f([-1.0705269641866977]) = 0.004974
>1, new best f([-0.9332442167834278]) = 0.004456
>1, new best f([-1.022829583444282]) = 0.000521
>6, new best f([-0.9958642058435989]) = 0.000017
>14, new best f([-0.9989132354583119]) = 0.000001
>40, new best f([-0.999748411998941]) = 0.000000
Done!
f([-0.999748411998941]) = 0.000000
```

Program 2: Particle Swarm Optimization for Function Optimization

Problem Statement: Implement the PSO algorithm using Python to optimize a mathematical function.

Algorithm:

7/11/24

Algorithm 2 : Particle Swarm Optimization

Fitness - function (x_1, x_2) :

$f_1 \leftarrow x1 - 2(x2) + 3$

$f_2 \leftarrow 2x1 + x2 - 8$

$z \leftarrow f1^2 + f2^2$

return z

Update - velocity (particle, velocity , pbest , gbest) :

Inertia

$\omega \leftarrow$ random . uniform (w-min, max)

Social & Cognitive components

$c_1 \leftarrow c$

$c_2 \leftarrow c$

$r_1 \leftarrow$ random . uniform (0 , max)

$r_2 \leftarrow$ random . uniform (0 , max)

new - velocity $\leftarrow$ [0 ..... num - particle]

for $i \leftarrow 0$ to num particle

new - velocity [i] $\leftarrow$ w * velocity [i] + c1 * r1 * (pbest [i] - particle [i]) + c2 * r2 * (gbest [i] - particle [i])

return new - velocity

Update - position (particle , velocity) :

new - particle $\leftarrow$ particle + velocity

return new - particle

pso_2d (population, dimension, pos_min, pos_max, generation, fitness_criteria):

```
particles ← [[random.uniform(pos_min, pos_max)
        for j ← 0 to dimension] for
            i ← 0 to population]

pbest_pos ← particles
pbest_fitness ← fitness_function (p[0], p[1])
                    for all p in particles

gbest_index ← min (pbest_fitness)
gbest_pos ← pbest_pos [gbest_index]
velocity ← [0... dimension] for i in range of
                    population

for t ← 0 to generation
    if average (pbest_fitness) <= fitness_criteria
        break
    else
        for n ← 0 to population
            velocity[n] ← update_velocity (
                    particles[n], velocity[n],
                    pbest_pos[n], gbest_pos)

            particles[n] ← update_position (
                    particles[n], velocity[n])

        pbest_fitness ← fitness_function (p[0], p[1])
        gbest_index ← min (pbest_fitness)
        gbest_pos ← pbest_pos [gbest_index]
```

Inputs :

population = 100

dimension = 2

pos_min = -100.0

pos_max = 100

generation = 100

fitness_criterion = 10e-6

SSO
7/11/24.

Code:

```
import random
import numpy as np
from matplotlib import pyplot as plt
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):
  # Initialise new velocity array
  num_particle = len(particle)
  new_velocity = np.array([0.0 for i in range(num_particle)])
  # Randomly generate r1, r2 and inertia weight from normal distribution
  r1 = random.uniform(0,max)
  r2 = random.uniform(0,max)
  w = random.uniform(w_min,max)
  c1 = c
  c2 = c
  # Calculate new velocity
  for i in range(num_particle):
    new_velocity[i] = w*velocity[i] + c1*r1*(pbest[i]-particle[i])+c2*r2*(gbest[i]-particle[i])
  return new_velocity
```

```
def update_position(particle, velocity):
  # Move particles by adding velocity
  new_particle = particle + velocity
  return new_particle



def pso_2d(population, dimension, position_min, position_max, generation, fitness_criterion):
  # Initialisation
  # Population
  particles = [[random.uniform(position_min, position_max) for j in range(dimension)] for i in
range(population)]
  # Particle's best position
  pbest_position = particles
  # Fitness
  pbest_fitness = [fitness_function(p[0],p[1]) for p in particles]
  # Index of the best particle
  gbest_index = np.argmin(pbest_fitness)
  # Global best particle position
  gbest_position = pbest_position[gbest_index]
  # Velocity (starting from 0 speed)
  velocity = [[0.0 for j in range(dimension)] for i in range(population)]


  # Loop for the number of generation
  for t in range(generation):
    # Stop if the average fitness value reached a predefined success criterion
    if np.average(pbest_fitness) <= fitness_criterion:
      break
    else:
      for n in range(population):
        # Update the velocity of each particle
        velocity[n] = update_velocity(particles[n], velocity[n], pbest_position[n], gbest_position)
        # Move the particles to new position
        particles[n] = update_position(particles[n], velocity[n])
    # Calculate the fitness value
    pbest_fitness = [fitness_function(p[0],p[1]) for p in particles]
    # Find the index of the best particle
```

```
    gbest_index = np.argmin(pbest_fitness)
    # Update the position of the best particle
    gbest_position = pbest_position[gbest_index]


  # 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 Generation: ', t)


population = 100
dimension = 2
position_min = -100.0
position_max = 100.0
generation = 100
fitness_criterion = 10e-6


pso_2d(population, dimension, position_min, position_max, generation, fitness_criterion)
```

Output:

```
Global Best Position:  [2.6000003  2.80000327]
Best Fitness Value:  5.3854415182325324e-11
Average Particle Best Fitness Value:  8.509552783246299e-06
Number of Generation:  86
```

Program 3: Ant Colony Optimization for the Traveling Salesman Problem

Problem Statement: Implement the ACO algorithm using Python to solve the TSP, where the objective is to find the shortest possible route that visits a list of cities and returns to the origin city.

Algorithm:

14/11/24

Algorithm 3 Ant Colony Optimization for TSP

```
Inputcity ( ) :
    n ← no of cities
    cities ← list of city
    for i ← 0 to n
        input city [i] x,y coords.

    return cities

Distance (p1, p2) :
    return sqrt( sum( (p1 - p2) ** 2 ) )

Build _ Path ( n, pheromone, points , α , β ) :
    visited ← for i ← 0 to n all false
    visited [current] ← true
    current ← random( n-points )
    path ← [current]
    path - length ← 0

    while false in visited :
        probabilities ← [0 .... n] all zeros array
        unvisited ← find point which is not visited
                    using array

        for i , unvisited - point in unvisited :
            pherone - value ← pherone [current, unvisited point]
                                  ** α

            distance - value ← distance (points [current],
                         points [unvisited - point] ) ** β

            probabilities [i] = pherone _ value / distance - value
```

Sum and normalise probabi...

next_point ← random (unvisited , p = probabilities

path. append (next - point)

path - length += distance (points [current - point]

points[next - point])

visited [next - point] ← True

current - point ← next - point

return path, path_length

Update - pheromones (pheromone , paths , path - lengths ,

rho, Q) :

pheromone ← pheromone * rho

for path , path - length in their arrays :

for i ← 0 to n-1

pheromone [path[i] , path[i+1]] ← phe...

+ Q/path_len...

pheromone [path[-1], path[0]] += Q / path-length

return pheromone

Ant - Colony - Optimisation (points, n-ants , n , α , β , rho ...

n - points ← len(points)

pheromone ← [(1, 1), (1, 1) . . . .] for n-points

best - path - length ← ∞

for iter in range of n :

paths ← [ ]

path - lengths ← [ ]

for _ in range of n_ants :
path, path_length ← Build_path (n, pheromone, α, β, points)

paths. append (path)

path_lengths . append (path_length)

if path_length < best_path_length

best_path ← path

best_path_length ← path_length

pheromone ← update_pheromones (pheromones, paths, path_lengths, rho, Q)

return best_path , best_path_length

points ← input_city ( )

Inputs:

points,

n_ants = 10

n_iters = 100

α = 1

β = 1

rho = 0.5

Q = 1

14/11/24.

Code:

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

```
# 1. Define the Problem: Taking custom 2D city coordinates as input
def input_city_coordinates():
    """
    Function to input custom city coordinates.
    The user is prompted to input coordinates for each city.
    """
    n_cities = int(input("Enter the number of cities: "))
    cities = []


    for i in range(n_cities):
        # Taking x and y coordinates as input
        x, y = map(float, input(f"Enter coordinates for city {i + 1} (x, y): ").split())
        cities.append([x, y])


    return np.array(cities)  # Return as a NumPy array for convenience


# 2. Distance Function: Calculate Euclidean distance between two cities
def distance(point1, point2):
    return np.sqrt(np.sum((point1 - point2)**2))


# 3. Construct Solutions: Build a solution for each ant
def construct_solution(n_points, pheromone, points, alpha, beta):
    visited = [False] * n_points
    current_point = np.random.randint(n_points)  # Start from a random city
    visited[current_point] = True
    path = [current_point]
    path_length = 0


    while False in visited:
        unvisited = np.where(np.logical_not(visited))[0]
        probabilities = np.zeros(len(unvisited))


        # Calculate the probabilities for the unvisited cities
        for i, unvisited_point in enumerate(unvisited):
            pheromone_value = pheromone[current_point, unvisited_point] ** alpha
            distance_value = distance(points[current_point], points[unvisited_point]) ** beta
            probabilities[i] = pheromone_value / distance_value
```

```
            # Normalize the probabilities
            probabilities /= np.sum(probabilities)


            # Choose the next city based on probabilities
            next_point = np.random.choice(unvisited, p=probabilities)
            path.append(next_point)
            path_length += distance(points[current_point], points[next_point])
            visited[next_point] = True
            current_point = next_point


        return path, path_length


    # 4. Update Pheromones: Update pheromone levels based on the ants' solutions
    def update_pheromones(pheromone, paths, path_lengths, evaporation_rate, Q):
        pheromone *= evaporation_rate  # Evaporate all pheromones
        for path, path_length in zip(paths, path_lengths):
            for i in range(len(path) - 1):
                pheromone[path[i], path[i + 1]] += Q / path_length
            pheromone[path[-1], path[0]] += Q / path_length  # Close the loop
        return pheromone


    # 5. Main ACO Algorithm: Main function to run the ACO
    def ant_colony_optimization(points, n_ants, n_iterations, alpha, beta, evaporation_rate, Q):
        n_points = len(points)
        pheromone = np.ones((n_points, n_points))  # Initial pheromone levels
        best_path = None
        best_path_length = np.inf


        # 6. Iterate: Run the ACO for a set number of iterations
        for iteration in range(n_iterations):
            paths = []
            path_lengths = []


            # Construct solutions for each ant
            for _ in range(n_ants):
```

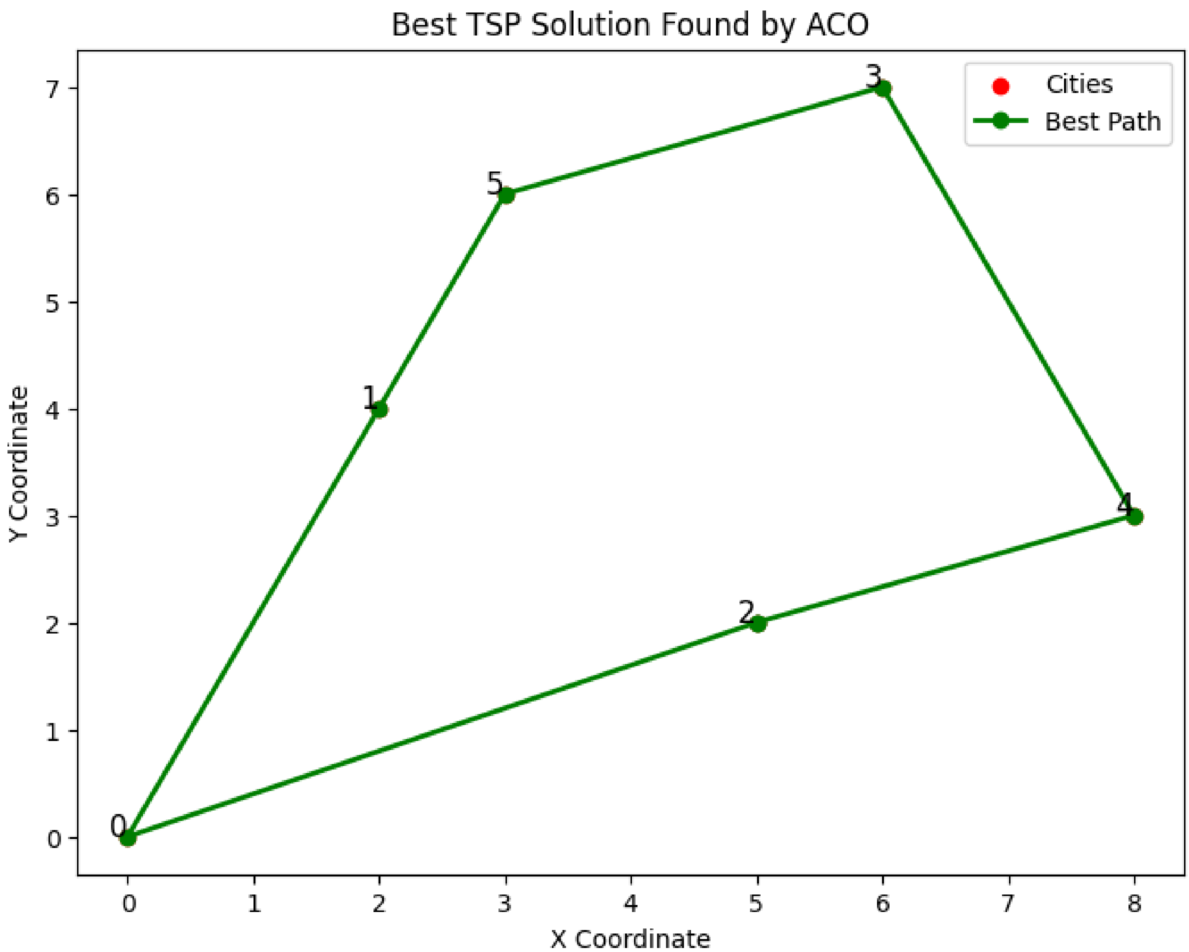
```
            path, path_length = construct_solution(n_points, pheromone, points, alpha, beta)
            paths.append(path)
            path_lengths.append(path_length)


            # Update the best solution found
            if path_length < best_path_length:
                best_path = path
                best_path_length = path_length


        # Update pheromones based on all ants' paths
        pheromone = update_pheromones(pheromone, paths, path_lengths, evaporation_rate, Q)


        # Optional: Print or log progress
        print(f"Iteration {iteration + 1}/{n_iterations}, Best Path Length: {best_path_length:.2f}")


    # Return the best path found
    return best_path, best_path_length


# 7. Output the Best Solution: Plotting the best path in 2D space
def plot_best_path(points, best_path):
    fig, ax = plt.subplots(figsize=(8, 6))
    ax.scatter(points[:, 0], points[:, 1], c='red', marker='o', label='Cities')


    # Draw the best path found by the ants
    path_points = points[best_path]
    path_points = np.vstack([path_points, path_points[0]])  # Close the loop
    ax.plot(path_points[:, 0], path_points[:, 1], c='green', linewidth=2, marker='o', label='Best
Path')


    # Display the cities' indices
    for i, point in enumerate(points):
        ax.text(point[0], point[1], str(i), fontsize=12, ha='right')


    ax.set_xlabel('X Coordinate')
    ax.set_ylabel('Y Coordinate')
```

```
    ax.set_title('Best TSP Solution Found by ACO')
    ax.legend()
    plt.show()


# Example usage with custom city inputs:
points = input_city_coordinates()  # Take custom input for city coordinates
best_path, best_path_length = ant_colony_optimization(
    points,
    n_ants=10,            # Number of ants
    n_iterations=100,     # Number of iterations
    alpha=1,              # Pheromone importance
    beta=1,               # Distance importance
    evaporation_rate=0.5, # Evaporation rate
    Q=1                   # Pheromone deposit
)


print(f"Best Path: {best_path}")
print(f"Best Path Length: {best_path_length:.2f}")


# Plot the best path found
plot_best_path(points, best_path)
```

Output:

```
Enter the number of cities: 6
Enter coordinates for city 1 (x, y): 0 0
Enter coordinates for city 2 (x, y): 2 4
Enter coordinates for city 3 (x, y): 5 2
Enter coordinates for city 4 (x, y): 6 7
Enter coordinates for city 5 (x, y): 8 3
Enter coordinates for city 6 (x, y): 3 6
Iteration 1/100, Best Path Length: 18.86
Iteration 2/100, Best Path Length: 18.86
Iteration 3/100, Best Path Length: 18.86
Iteration 4/100, Best Path Length: 18.86
Iteration 5/100, Best Path Length: 18.86
Iteration 6/100, Best Path Length: 18.86
Iteration 7/100, Best Path Length: 18.86
Iteration 8/100, Best Path Length: 18.86
Iteration 9/100, Best Path Length: 17.50
Iteration 10/100, Best Path Length: 17.50
Iteration 11/100, Best Path Length: 17.50
Iteration 12/100, Best Path Length: 17.50
Iteration 13/100, Best Path Length: 17.50
Iteration 14/100, Best Path Length: 17.50
Iteration 15/100, Best Path Length: 17.50
Iteration 16/100, Best Path Length: 17.50
Iteration 17/100, Best Path Length: 17.50
Iteration 18/100, Best Path Length: 17.50
```

```
Iteration 92/100, Best Path Length: 17.50
Iteration 93/100, Best Path Length: 17.50
Iteration 94/100, Best Path Length: 17.50
Iteration 95/100, Best Path Length: 17.50
Iteration 96/100, Best Path Length: 17.50
Iteration 97/100, Best Path Length: 17.50
Iteration 98/100, Best Path Length: 17.50
Iteration 99/100, Best Path Length: 17.50
Iteration 100/100, Best Path Length: 17.50
Best Path: [2, 4, 3, 5, 1, 0]
Best Path Length: 17.50
```



Program 4: Cuckoo Search (CS)

Problem Statement: Implement cuckoo search algorithm to optimize a function using nests and eggs.

Algorithm:

21/11/24

Algorithm 4: Cuckoo Search Algorithm

```
objective_function(x):
    return x[0]**2

levy_flight(num, β=1.5):
    sigma_u ← ( gamma(1+β) * sin(π * β/2) / (gamma((1+β)/2) * β * 2**((β-1)/2)) )**(1/β)

    u ← random.normal(0, sigma_u, num)
    v ← random.normal(0, 1, num)

    return u / |v|^(1/β)


cuckoo_search(iter, num_nests, pa=0.25):
    num_dim ← 1
    nests ← random.rand(num_nests, num_dim)
    fitness ← objective_function(nests)
    best_nest ← nests[np.argmin(fitness)]
    best_fitness ← np.min(fitness)
    for _ in range(iter):
        for i ← 0 to num_nests:
            new_nest ← nests[i] + levy_flight(num_d...)

            new_fitness ← objective_function(new_nest)

            if new_fitness < fitness[i]:
                nests[i] ← new_nest
                fitness[i] ← new_fitness
```

worst_nests ← argsort (fitness) [- int(pa * num_nests):]
for j in worst_nests:
    nests[j] ← random (num_dim) * 10 - 5
    fitness[j] ← objective_function (nests[j])

Current_best ← np.argmin (fitness)
current_best_fitness ← fitness [current_best]
if current_best_fitness < best_fitness:
    best_fitness ← current_best_fitness
    best_nest ← nests [current_best]

return best_nest, best_fitness

Inputs: num_iterations = 1000
num_nests = 25
pa = 0.25
β = 1.5

21/11/24

Code:

```
import numpy as np


# Objective function for 1D (x^2)
def objective_function_1d(x):
    return x[0]**2  # x is a 1D array, even though we just care about the first element


# Lévy Flight to generate new solutions
```

```
def levy_flight(num_dim, beta=1.5):
    sigma_u = (np.math.gamma(1 + beta) * np.sin(np.pi * beta / 2) /
            np.math.gamma((1 + beta) / 2) * beta * (2 ** ((beta - 1) / 2)))**(1 / beta)
    u = np.random.normal(0, sigma_u, num_dim)  # Lévy-distributed steps
    v = np.random.normal(0, 1, num_dim)
    return u / np.abs(v) ** (1 / beta)


# Cuckoo Search Algorithm for 1D
def cuckoo_search_1d(num_iterations, num_nests, pa=0.25):
    num_dim = 1  # 1D problem
    nests = np.random.rand(num_nests, num_dim) * 10 - 5  # Random initialization within [-5, 5]
    fitness = np.apply_along_axis(objective_function_1d, 1, nests)  # Evaluate initial fitness
    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)


    for _ in range(num_iterations):
        for i in range(num_nests):
            new_nest = nests[i] + levy_flight(num_dim)  # Generate new solution using Lévy flight
            new_fitness = objective_function_1d(new_nest)


            if new_fitness < fitness[i]:  # Replace if new solution is better
                nests[i] = new_nest
                fitness[i] = new_fitness


        # Abandon the worst nests
        worst_nests = np.argsort(fitness)[-int(pa * num_nests):]
        for j in worst_nests:
            nests[j] = np.random.rand(num_dim) * 10 - 5  # Randomly initialize new nests
            fitness[j] = objective_function_1d(nests[j])


        # Update best solution found so far
        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]
        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]
```

```
    return best_nest, best_fitness  # Return the best solution and its fitness


# Run the cuckoo search on the 1D problem
best_solution, best_fitness = cuckoo_search_1d(num_iterations=1000, num_nests=25)
print(f"Best solution found: {best_solution} with objective value: {best_fitness}")
```

Output:

```
Best solution found: [2.93269108e-05] with objective value: 8.600676942579324e-10
```

Program 5: Grey Wolf Optimizer (GWO)

Problem Statement: Implement grey wolf optimizer using alpha, beta and gamma wolves.

Algorithm:

28/11/24

Algorithm 5 : Grey Wolf Optimization

Initialize_wolves (search_space, num_wolves):
    dim ← len (search_space)
    wolves ← zeros ((num_wolves, dim))
    for i ← 0 to num_wolves:
        wolves [i] ← random.uniform (
                search_space [:, 0],
                search_space [:, 1])

    return wolves

Fitness_function (x):
    return np.sum (x**2)

GWO_algorithm (search_space, num_wolves,
                max_iterations):

    dim ← len (search_space)

    wolves ← initialise_wolves (search_space,
                num_wolves)

    alpha ← zeros (dim)
    beta ← zeros (dim)
    gamma ← zeros (dim)

    alpha_fit = beta_fit = gamma_fit ← float('i

    best_fit ← float ('inf')

```
for iter in range (max)
    a ← 2 - (iteration / max) * 2
    for i ← 0 to num_wolves:
        fitness ← fitness_function(wolves[i])
        if fitness < alpha_fitness:
            copy beta wolf & fitness to gamma
            copy alpha wolf & fitness to beta
            alpha_wolf ← wolves[i].copy()
            alpha_fitness ← fitness

        elif fitness < beta_fitness:
            copy beta wolf & fitness to gamma
            beta_wolf ← wolves[i].copy()
            beta_fitness ← fitness

        elif fitness < gamma_fitness:
            gamma_wolf ← wolves[i].copy()
            gamma_fitness ← fitness

    if alpha_fitness < best_fitness:
        best_fitness ← alpha_fitness

    for i ← 0 to num_wolves:
        for j ← 0 to dims:
            r1 & r2 random floats
            A1 ← 2*a*r1 - a
            C1 ← 2*r2
            D_alpha ← abs(C1 * alpha_wolf[j]
                                - wolves[i, j])

            X1 ← alpha_wolf[j] - A1 * D_alpha
```

New r1 & r2 random floats

A2 ← 2 * a * r1 − a

C2 ← 2 * r2

D-beta ← abs (C2 * beta-wolf [j]
- wolves [i, j])

x2 ← beta-wolf [j] − A2 * D-beta

New r1 & r2 random floats

A3 ← 2 * a * r1 − a

C3 ← 2 * r2

D-gamma ← abs (C3 * gamma-wolf [j]
- wolves [i, j])

x3 ← gamma-wolf [j] − A3 * D-gamma

wolves [i, j] ← (x1 + x2 + x3) / 3

wolves [i, j] ← clip (wolves [i, j] ,
search-space [j, 0] ,
search-space [j, 1])

Inputs:

search-space = array ([−5, 5], [−5, 5])

num-wolves = 10

max-iter = 100

Code:

```
import numpy as np

def initialize_wolves(search_space, num_wolves):
    dimensions = len(search_space)
    wolves = np.zeros((num_wolves, dimensions))
    for i in range(num_wolves):
        wolves[i] = np.random.uniform(search_space[:, 0], search_space[:, 1])
    return wolves

def fitness_function(x):
    # Define your fitness function to evaluate the quality of a solution
    # Example: Sphere function (minimize the sum of squares)
    return np.sum(x**2)

def gwo_algorithm(search_space, num_wolves, max_iterations):
    dimensions = len(search_space)

    # Initialize wolves
    wolves = initialize_wolves(search_space, num_wolves)

    # Initialize alpha, beta, and gamma wolves
    alpha_wolf = np.zeros(dimensions)
    beta_wolf = np.zeros(dimensions)
    gamma_wolf = np.zeros(dimensions)

    # Initialize the fitness of alpha, beta, gamma wolves
    alpha_fitness = float('inf')
    beta_fitness = float('inf')
    gamma_fitness = float('inf')

    # Store the best fitness found
    best_fitness = float('inf')

    for iteration in range(max_iterations):
        a = 2 - (iteration / max_iterations) * 2  # Parameter a decreases linearly from 2 to 0

        #print(f"Iteration {iteration + 1}/{max_iterations}")

        # Evaluate the fitness of all wolves
        for i in range(num_wolves):
            fitness = fitness_function(wolves[i])
```

```
        # Print the fitness of the current wolf
       # print(f"Wolf {i+1} Fitness: {fitness}")

        # Update alpha, beta, gamma wolves based on fitness
        if fitness < alpha_fitness:
            gamma_wolf = beta_wolf.copy()
            gamma_fitness = beta_fitness
            beta_wolf = alpha_wolf.copy()
            beta_fitness = alpha_fitness
            alpha_wolf = wolves[i].copy()
            alpha_fitness = fitness
        elif fitness < beta_fitness:
            gamma_wolf = beta_wolf.copy()
            gamma_fitness = beta_fitness
            beta_wolf = wolves[i].copy()
            beta_fitness = fitness
        elif fitness < gamma_fitness:
            gamma_wolf = wolves[i].copy()
            gamma_fitness = fitness

    # Print the best fitness for this iteration
    #print(f"Best Fitness in this Iteration: {alpha_fitness}")

    # Store the best overall fitness found so far
    if alpha_fitness < best_fitness:
        best_fitness = alpha_fitness

    # Update positions of wolves
    for i in range(num_wolves):
        for j in range(dimensions):
            r1 = np.random.random()
            r2 = np.random.random()

            A1 = 2 * a * r1 - a
            C1 = 2 * r2

            D_alpha = np.abs(C1 * alpha_wolf[j] - wolves[i, j])
            X1 = alpha_wolf[j] - A1 * D_alpha

            r1 = np.random.random()
            r2 = np.random.random()

            A2 = 2 * a * r1 - a
```

```
            C2 = 2 * r2

            D_beta = np.abs(C2 * beta_wolf[j] - wolves[i, j])
            X2 = beta_wolf[j] - A2 * D_beta

            r1 = np.random.random()
            r2 = np.random.random()

            A3 = 2 * a * r1 - a
            C3 = 2 * r2

            D_gamma = np.abs(C3 * gamma_wolf[j] - wolves[i, j])
            X3 = gamma_wolf[j] - A3 * D_gamma

            # Update the wolf's position
            wolves[i, j] = (X1 + X2 + X3) / 3

            # Ensure the new position is within the search space bounds
            wolves[i, j] = np.clip(wolves[i, j], search_space[j, 0], search_space[j, 1])

    print(f"Optimal Solution Found: {alpha_wolf}")
    print(f"Optimal Fitness: {best_fitness}")
    return alpha_wolf  # Return the best solution found

# Example usage
search_space = np.array([[-5, 5], [-5, 5]])  # Define the search space for the optimization problem
num_wolves = 10  # Number of wolves in the pack
max_iterations = 100  # Maximum number of iterations

# Run the GWO algorithm
optimal_solution = gwo_algorithm(search_space, num_wolves, max_iterations)

# Print the optimal solution
print("Optimal Solution:", optimal_solution)
```

Output:

```
Optimal Solution Found: [ 1.51778516e-13 -1.31752029e-13]
Optimal Fitness: 4.039531525040229e-26
Optimal Solution: [ 1.51778516e-13 -1.31752029e-13]
```

Program 6: Parallel Cellular Algorithms and Programs

Problem Statement: Implement the parallel cellular algorithm to optimize the function.

Algorithm:

Algorithm 6 Parallel Cellular Algorithms

custom - function (x) :
    return sum (x ** 2)

Initialize - population (grid - size , dim , lower, upper)
    population ← random (lower , upper , (grid - size ,
        grid-size , dim))
    return population

Evaluate - fitness (population , fitness _ function) :
    fitness ← zeros (population . shape [: -1])
    for i in range (population . Shape [0])
        for j in range (population . shape [1])
            fitness [i] [j] ← ~~fitne~~ custom - function
                (population [i , j])

    return fitness

update _ States (pop , fit , radius , lower , upper) :
    grid - size , _ , dim ← ~~population~~ . shape
    new - pop ← pop . copy ( )
    for i in 0 to grid - size :
        for j in 0 to grid - size :
            neighbours ← get - neighbors (i , j , grid - size ,
                radius)
            best - neighbor ← None
            best - fitness ← float (inf)

        for ni , nj in neighbors :
            if fitness [ni , nj] < best - fitness .
                best - fitness ← fitness [ni , nj]
                best - neighbor ← (ni , nj)

```
if best_neighbors:
    ni, nj ← best_neighbors
    new_population [i, j] ← pop[ni, nj] +
                            random(-0.5, 0.5, d
    new_population [i, j] ← clip(new_populat
                    [i, j], lower, upper)

return new_population

get_neighbors( i, j, grid_size, radius):
    neighbors ← [ ]
    for di in range (-radius, radius + 1):
        for dj in range (-radius, radius + 1):
            ni, nj ← (i + di) % grid_size,
                     (j + dj) % grid_size

            if (di != 0 or dj != 0):
                neighbors.append ((ni, nj))

    return neighbors

Parallel_cellular_Algorithm (grid_size, dim, lower,
    upper, max_iters, radius, fitness_func).
        population ← initialise_population (grid_size,
                        dim, lower, upper)

        fitness ← evaluate_fitness (population,
                        fitness_function)

        min_fitness ← fitness.min( )
        if min_fitness < best_fitness
            best_fitness ← min_fitness
```

```
best_sol ← None
best_fit ← ∞

for iter in max_iters :
        pop ← update_states ( population, fitness,
              radius, lower, upper)

        fitness ← evaluate_fitness ( pop,
                  fitness_function)

        min_fitness ← fitness.min( )
        if min_fitness < best_fitness :
            best_fitness ← min_fitness
            best_sol ← population [np.
                 unravel_index ( fitness.min( ),
                 fitness.shape )]

        print (iteration , fitness)

return best_sol, best_fit
```

Inputs:
grid_size = 10
dim = 1
lower = -5
upper = 5
max_iters = 100
radius = 1

Code:

```
import numpy as np
import random

# Define any optimization function to minimize (can be changed as needed)
def custom_function(x):
    return x**2  # Example function: x^2 to minimize

# Initialize the population (grid of cells) with random values
def initialize_population(grid_size, dim, lower_bound, upper_bound):
    # Initialize a grid of cells with random positions
    population = np.random.uniform(lower_bound, upper_bound, (grid_size, grid_size, dim))
    return population

# Evaluate the fitness of each cell in the grid
def evaluate_fitness(population, fitness_function):
    # Calculate fitness of each cell based on the optimization function
    fitness = np.zeros(population.shape[:-1])  # Create an empty fitness matrix
    for i in range(population.shape[0]):
        for j in range(population.shape[1]):
            fitness[i, j] = fitness_function(population[i, j])  # Assign fitness value
    return fitness

# Update the state of each cell based on the best neighbor within a neighborhood
def update_states(population, fitness, neighborhood_radius, lower_bound, upper_bound):
    grid_size, _, dim = population.shape
    new_population = population.copy()  # Make a copy to store updated values

    # Iterate over each cell in the grid
    for i in range(grid_size):
        for j in range(grid_size):
            # Get the neighbors of the current cell
            neighbors = get_neighbors(i, j, grid_size, neighborhood_radius)
            best_neighbor = None
            best_fitness = float('inf')

            # Find the best neighbor based on fitness
            for ni, nj in neighbors:
                if fitness[ni, nj] < best_fitness:
                    best_fitness = fitness[ni, nj]
                    best_neighbor = (ni, nj)

            # Update the current cell towards the best neighbor
```

```
            if best_neighbor:
                ni, nj = best_neighbor
                # Add a small random perturbation to the updated cell value
                new_population[i, j] = population[ni, nj] + np.random.uniform(-0.5, 0.5, dim)
                # Ensure the new population is within bounds
                new_population[i, j] = np.clip(new_population[i, j], lower_bound, upper_bound)

    return new_population

# Get the neighbors of a cell within a given neighborhood radius
def get_neighbors(i, j, grid_size, radius):
    neighbors = []
    for di in range(-radius, radius + 1):
        for dj in range(-radius, radius + 1):
            ni, nj = (i + di) % grid_size, (j + dj) % grid_size
            if (di != 0 or dj != 0):  # Exclude the cell itself
                neighbors.append((ni, nj))
    return neighbors

# Main function to run the parallel cellular algorithm
def parallel_cellular_algorithm(grid_size, dim, lower_bound, upper_bound, max_iterations,
neighborhood_radius, fitness_function):
    # Initialize the population (grid of cells)
    population = initialize_population(grid_size, dim, lower_bound, upper_bound)

    # Initialize fitness of the population
    fitness = evaluate_fitness(population, fitness_function)

    best_solution = None
    best_fitness = float('inf')

    # Iterate to update states (based on number of iterations)
    for iteration in range(max_iterations):
        # Update states based on neighbor interactions
        population = update_states(population, fitness, neighborhood_radius, lower_bound,
upper_bound)

        # Re-evaluate fitness
        fitness = evaluate_fitness(population, fitness_function)

        # Track the best solution found so far
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
```

```
            best_fitness = min_fitness
            best_solution = population[np.unravel_index(fitness.argmin(), fitness.shape)]

        # Print output every 10 iterations
        if (iteration + 1) % 10 == 0:
            print(f"Iteration {iteration + 1}/{max_iterations}, Best Fitness: {best_fitness}")

    return best_solution, best_fitness

# Parameters for the algorithm
grid_size = 10  # Number of cells per side (10x10 grid = 100 cells)
dim = 1  # One-dimensional solution space (this can be extended to higher dimensions if needed)
lower_bound = -5  # Lower bound for the solution space (can be adjusted for different problem
ranges)
upper_bound = 5  # Upper bound for the solution space
max_iterations = 100  # Number of iterations (how long to run the algorithm)
neighborhood_radius = 1  # Radius for neighborhood search (defines how far neighboring cells
are considered)

# Run the parallel cellular algorithm
best_solution, best_fitness = parallel_cellular_algorithm(grid_size, dim, lower_bound,
upper_bound, max_iterations, neighborhood_radius, custom_function)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_fitness)
```

Output:

```
Iteration 10/100, Best Fitness: 6.1517480231833636e-06
Iteration 20/100, Best Fitness: 1.6757075383394441e-07
Iteration 30/100, Best Fitness: 5.953459053884453 4e-08
Iteration 40/100, Best Fitness: 5.9534590538844534e-08
Iteration 50/100, Best Fitness: 5.9534590538844534e-08
Iteration 60/100, Best Fitness: 5.9534590538844534e-08
Iteration 70/100, Best Fitness: 5.9534590538844534e-08
Iteration 80/100, Best Fitness: 5.9534590538844534e-08
Iteration 90/100, Best Fitness: 5.9534590538844534e-08
Iteration 100/100, Best Fitness: 5.3029915870773e-10

Best Solution Found: [2.30282253e-05]
Best Fitness Value: 5.3029915870773e-10
```

Program 7:Optimization via Gene Expression Algorithms

Problem Statement: Implement gene expression to optimize the math function using crossover and mutation rates.

Algorithm:

Algorithm 7 : Optimization via Gene Expression

Function (x):
    return sum (x**2)

Initialize_Population (pop_size, num_genes, lower, upper
    pop ← random (lower, upper, (pop_size, num_genes
    return pop

Evaluate_fitness (pop, ~~fitness~~ function):
    fitness ← zeros (pop.shape [0])
    for i in range pop.shape [0] :
        fitness [i] ← ~~fitness~~ function (pop [i])
    return fitness

Selection (pop , fitness , num_selected):
    prob ← fitness / fitness.sum()
    indices ← random.choice (range (len (pop)),
            size = num_selected , p = prob)
    selected_pop ← pop [indices]
    return selected_pop

Crossover (selected_pop , crossover):
    new_pop ← [ ]
    num_individuals ← len (selected_pop)
    for i in range 0 to num_individuals - 1 , step 2 :
        p1 , p2 ← selected_pop [i] , selected_pop [i+1]
        if len (p1) > 1 and random < crossover :
            crossover_point ← random (1, len (p1) -
            child 1 ← concatenate (p1 [: crossover_point
                p2 [crossover_point :])

```
        child2 ← concatenate (p2[: crossover_point ],
                     p1[ crossover_point : ])
        new_pop. extend([p1, p2])

    if num_individuals % 2 == 1 :
        new_pop append (selected_pop [-1])

    return new_pop

Mutation (pop, rate, lower, upper) :
    for i in range (pop.shape[0]) :
        if random < rate :
            gene ← random (0, pop.shape[1] - 1)
            pop[i, gene] ← random_uniform (lower,
                                            upper)

    .
    return pop

Gene_expression (individual, function) :
    return function (individual)

Gene_Expression_Algorithm (pop_size, num_genes, lower, upper,
  max_gen, mutation_rate, crossover_rate, function) :
    pop ← initialise_population (pop_size, num_genes,
             lower, upper)

    ~~best ← ~~
    best_sol ← NIL
    best_fit ← ∞

    for gen in 0 to max_gen :
        fitness ← evaluate_fitness (pop, function)
        min_fitness ← fitness.min()
        if min_fitness < best_fit :
            best_fit ← min_fitness
            best_sol ← pop [min(fitness)]

        selected_pop ← selection (pop, fitness,
                         pop_size // 2)
```

offspring-pop ← crossover (selected-pop
crossover-rate)

pop ← mutation (offspring-pop, mutation,
lower, upper)

print (Generation, fitness)

return best-sol, best-fit

Inputs:

pop-size = 50
num-genes = 1
lower = -5
upper = 5
max-gen = 100
mutation-rate = 0.1
crossover_rate = 0.7

19/12/24

Code:

```
import numpy as np
import random

# Define any optimization function to minimize (can be changed as needed)
def custom_function(x):
    # Example function: x^2 to minimize
    return np.sum(x ** 2)  # Ensuring the function works for multidimensional inputs
```

```
# Initialize population of genetic sequences (each individual is a sequence of genes)
def initialize_population(population_size, num_genes, lower_bound, upper_bound):
    # Create a population of random genetic sequences
    population = np.random.uniform(lower_bound, upper_bound, (population_size, num_genes))
    return population

# Evaluate the fitness of each individual (genetic sequence) in the population
def evaluate_fitness(population, fitness_function):
    fitness = np.zeros(population.shape[0])
    for i in range(population.shape[0]):
        fitness[i] = fitness_function(population[i])  # Apply the fitness function to each individual
    return fitness

# Perform selection: Choose individuals based on their fitness (roulette wheel selection)
def selection(population, fitness, num_selected):
    # Select individuals based on their fitness (higher fitness, more likely to be selected)
    probabilities = fitness / fitness.sum()  # Normalize fitness to create selection probabilities
    selected_indices = np.random.choice(range(len(population)), size=num_selected,
p=probabilities)
    selected_population = population[selected_indices]
    return selected_population

# Perform crossover: Combine pairs of individuals to create offspring
def crossover(selected_population, crossover_rate):
    new_population = []
    num_individuals = len(selected_population)

    for i in range(0, num_individuals - 1, 2):  # Iterate in steps of 2, skipping the last one if odd
        parent1, parent2 = selected_population[i], selected_population[i + 1]
        if len(parent1) > 1 and random.random() < crossover_rate:  # Only perform crossover if
more than 1 gene
            crossover_point = random.randint(1, len(parent1) - 1)  # Choose a random crossover
point
            offspring1 = np.concatenate((parent1[:crossover_point], parent2[crossover_point:]))
            offspring2 = np.concatenate((parent2[:crossover_point], parent1[crossover_point:]))
            new_population.extend([offspring1, offspring2])  # Create two offspring
        else:
            new_population.extend([parent1, parent2])  # No crossover, retain the parents

    # If the number of individuals is odd, carry the last individual without crossover
    if num_individuals % 2 == 1:
        new_population.append(selected_population[-1])
```

```
    return np.array(new_population)

# Perform mutation: Introduce random changes in offspring
def mutation(population, mutation_rate, lower_bound, upper_bound):
    for i in range(population.shape[0]):
        if random.random() < mutation_rate:  # Apply mutation based on the rate
            gene_to_mutate = random.randint(0, population.shape[1] - 1)  # Choose a random gene to
mutate
            population[i, gene_to_mutate] = np.random.uniform(lower_bound, upper_bound)  #
Mutate the gene
    return population

# Gene expression: In this context, it is how we decode the genetic sequence into a solution
def gene_expression(individual, fitness_function):
    return fitness_function(individual)

# Main function to run the Gene Expression Algorithm
def gene_expression_algorithm(population_size, num_genes, lower_bound, upper_bound,
max_generations, mutation_rate, crossover_rate, fitness_function):
    # Step 2: Initialize the population of genetic sequences
    population = initialize_population(population_size, num_genes, lower_bound, upper_bound)

    best_solution = None
    best_fitness = float('inf')

    # Step 9: Iterate for the specified number of generations
    for generation in range(max_generations):
        # Step 4: Evaluate fitness of the current population
        fitness = evaluate_fitness(population, fitness_function)

        # Track the best solution found so far
        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.argmin(fitness)]

        # Step 5: Perform selection (choose individuals based on fitness)
        selected_population = selection(population, fitness, population_size // 2)  # Select half of
the population

        # Step 6: Perform crossover to generate new individuals
        offspring_population = crossover(selected_population, crossover_rate)
```

```
        # Step 7: Perform mutation on the offspring population
        population = mutation(offspring_population, mutation_rate, lower_bound, upper_bound)

        # Print output every 10 generations
        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}/{max_generations}, Best Fitness: {best_fitness}")

    # Step 10: Output the best solution found
    return best_solution, best_fitness

# Parameters for the algorithm
population_size = 50  # Number of individuals in the population
num_genes = 1  # Number of genes (for a 1D problem, this is just 1, extendable for higher
dimensions)
lower_bound = -5  # Lower bound for the solution space
upper_bound = 5  # Upper bound for the solution space
max_generations = 100  # Number of generations to evolve the population
mutation_rate = 0.1  # Mutation rate (probability of mutation per gene)
crossover_rate = 0.7  # Crossover rate (probability of crossover between two parents)

# Run the Gene Expression Algorithm
best_solution, best_fitness = gene_expression_algorithm(
    population_size, num_genes, lower_bound, upper_bound,
    max_generations, mutation_rate, crossover_rate, custom_function)

# Output the best solution found
print("\nBest Solution Found:", best_solution)
print("Best Fitness Value:", best_fitness)
```

Output:

```
Generation 10/100, Best Fitness: 0.02125657210893126
Generation 20/100, Best Fitness: 0.020266700998504673
Generation 30/100, Best Fitness: 0.020266700998504673
Generation 40/100, Best Fitness: 0.0065667627493710655
Generation 50/100, Best Fitness: 0.00089546902711783
Generation 60/100, Best Fitness: 0.00089546902711783
Generation 70/100, Best Fitness: 0.0006225317320463783
Generation 80/100, Best Fitness: 0.0006225317320463783
Generation 90/100, Best Fitness: 0.0006225317320463783
Generation 100/100, Best Fitness: 0.0006225317320463783

Best Solution Found: [-0.02495059]
Best Fitness Value: 0.0006225317320463783
```