Bio Inspired Systems

Name: Sanghamitra R
USN: 1BM22CS237

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: 1.9980464857154846e-05
Generation 20/100, Best Fitness: 8.798327298656325e-06
Generation 30/100, Best Fitness: 8.798327298656325e-06
Generation 40/100, Best Fitness: 8.798327298656325e-06
Generation 50/100, Best Fitness: 8.798327298656325e-06
Generation 60/100, Best Fitness: 8.798327298656325e-06
Generation 70/100, Best Fitness: 8.798327298656325e-06
Generation 80/100, Best Fitness: 8.798327298656325e-06
Generation 90/100, Best Fitness: 8.798327298656325e-06
Generation 100/100, Best Fitness: 8.798327298656325e-06

Best Solution Found: [0.0029662]
Best Fitness Value: 8.798327298656325e-06
```

Application Problem

Question: Use GEP to model and predict environmental systems(pollution levels, climate change)by optimising parameters and equations based on historical data and simulation results.

Code:

```
import numpy as np
import random
import math
```

```
import pandas as pd
from sklearn.metrics import mean_squared_error

# Define the objective function based on the environment data (pollution,
temperature, etc.)
def environmental_objective(individual, data):
    """
    The objective function for environmental prediction.
    It evaluates how well the model (equation) represented by 'individual'
predicts actual data.
    """
    predictions = []
    for row in data:
        prediction = evaluate_equation(individual, row)  # Evaluate the
individual (equation)
        predictions.append(prediction)

    # Calculate the fitness (mean squared error between predictions and
actual data)
    true_values = data[:, -1]  # Assuming last column is the actual value
    mse = mean_squared_error(true_values, predictions)
    return mse  # Lower MSE is better

def evaluate_equation(individual, row):
    """
    Evaluate the equation represented by the individual (a sequence of
genes).
    'individual' will be a mathematical expression encoded in genes.
    """
    result = 0
    for i, gene in enumerate(individual):
        result += gene * row[i]  # A simple linear model: gene *
feature_value

    return result

# Initialize the population of genetic sequences (individuals)
def initialize_population(population_size, num_genes, lower_bound,
upper_bound):
```

```
    population = np.random.uniform(lower_bound, upper_bound,
(population_size, num_genes))
    return population

# Perform roulette wheel selection
def selection(population, fitness, num_selected):
    probabilities = fitness / fitness.sum()
    selected_indices = np.random.choice(range(len(population)),
size=num_selected, p=probabilities)
    selected_population = population[selected_indices]
    return selected_population

# Perform crossover to generate offspring
def crossover(selected_population, crossover_rate):
    new_population = []
    num_individuals = len(selected_population)

    for i in range(0, num_individuals - 1, 2):
        parent1, parent2 = selected_population[i], selected_population[i +
1]
        if random.random() < crossover_rate:
            crossover_point = random.randint(1, len(parent1) - 1)
            offspring1 = np.concatenate((parent1[:crossover_point],
parent2[crossover_point:]))
            offspring2 = np.concatenate((parent2[:crossover_point],
parent1[crossover_point:]))
            new_population.extend([offspring1, offspring2])
        else:
            new_population.extend([parent1, parent2])

    if num_individuals % 2 == 1:
        new_population.append(selected_population[-1])

    return np.array(new_population)

# Perform mutation: introduce random changes
def mutation(population, mutation_rate, lower_bound, upper_bound):
    for i in range(population.shape[0]):
        if random.random() < mutation_rate:
            gene_to_mutate = random.randint(0, population.shape[1] - 1)
```

```
            population[i, gene_to_mutate] = np.random.uniform(lower_bound,
upper_bound)
    return population

# Main function to run the Gene Expression Algorithm (GEP)
def gene_expression_algorithm(population_size, num_genes, lower_bound,
upper_bound, max_generations, mutation_rate, crossover_rate, data):
    population = initialize_population(population_size, num_genes,
lower_bound, upper_bound)

    best_solution = None
    best_fitness = float('inf')

    # Iterate over generations
    for generation in range(max_generations):
        fitness = np.array([environmental_objective(individual, data) for
individual in population])

        min_fitness = fitness.min()
        if min_fitness < best_fitness:
            best_fitness = min_fitness
            best_solution = population[np.argmin(fitness)]

        # Select individuals based on fitness
        selected_population = selection(population, fitness,
population_size // 2)

        # Perform crossover
        offspring_population = crossover(selected_population,
crossover_rate)

        # Perform mutation
        population = mutation(offspring_population, mutation_rate,
lower_bound, upper_bound)

        if (generation + 1) % 10 == 0:
            print(f"Generation {generation + 1}/{max_generations}, Best
Fitness (MSE): {best_fitness:.6f}")

    return best_solution, best_fitness
```

```
# Sample data: Replace this with actual environmental data (pollution levels, temperature, etc.)
# Each row represents a data point with features and the last column is the target value (e.g., pollution level)
data = np.array([
    [1.0, 2.0, 3.0, 4.0, 5.0, 15.0],  # Example row (features followed by actual value)
    [2.0, 3.0, 4.0, 5.0, 6.0, 20.0],
    [3.0, 4.0, 5.0, 6.0, 7.0, 25.0],
    [4.0, 5.0, 6.0, 7.0, 8.0, 30.0],
    [5.0, 6.0, 7.0, 8.0, 9.0, 35.0],
    [6.0, 7.0, 8.0, 9.0, 10.0, 40.0]
])

# Parameters for the GEP algorithm
population_size = 50
num_genes = 5  # Number of genes (equation components)
lower_bound = -10  # Lower bound for the solution space
upper_bound = 10   # Upper bound for the solution space
max_generations = 100  # Number of generations to evolve the population
mutation_rate = 0.1  # Mutation rate
crossover_rate = 0.7  # Crossover rate

# 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, data)

print("\nBest Solution Found:", best_solution)
print("Best Fitness (MSE):", best_fitness)
```

Output:

```
Generation 10/100, Best Fitness (MSE): 11.422478
Generation 20/100, Best Fitness (MSE): 11.422478
Generation 30/100, Best Fitness (MSE): 11.422478
Generation 40/100, Best Fitness (MSE): 11.422478
Generation 50/100, Best Fitness (MSE): 11.422478
Generation 60/100, Best Fitness (MSE): 11.422478
Generation 70/100, Best Fitness (MSE): 11.422478
Generation 80/100, Best Fitness (MSE): 11.422478
Generation 90/100, Best Fitness (MSE): 11.422478
Generation 100/100, Best Fitness (MSE): 11.422478

Best Solution Found: [ 5.51001406 -3.91428818  7.50820505 -9.77928146  5.96757112]
Best Fitness (MSE): 11.42247784545849
```