BIS LAB CIE

1BM22CS260
Shlok Shivaram Iyer
Section 5E

GENE EXPRESSION ALGORITHM

Code:

```python
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
```

```python
# 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)
```

```python
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: 0.06056864822640436
Generation 20/100, Best Fitness: 0.011778082415474256
Generation 30/100, Best Fitness: 0.011778082415474256
Generation 40/100, Best Fitness: 0.007493912919747368
Generation 50/100, Best Fitness: 0.007493912919747368
Generation 60/100, Best Fitness: 0.007493912919747368
Generation 70/100, Best Fitness: 0.004973267669799166
Generation 80/100, Best Fitness: 0.0001056233965407511
Generation 90/100, Best Fitness: 0.0001056233965407511
Generation 100/100, Best Fitness: 0.0001056233965407511

Best Solution Found: [-0.01027732]
Best Fitness Value: 0.0001056233965407511
```

Apply GEP to optimize the parameters and the algorithms for image recognition,enabling the accurate classification of images based on features such as color, texture and shape

Code:

```
import numpy as np
import random

# Define the Sphere function as the fitness
function
def sphere_function(individual, color, texture,
shape, labels):
    """
    Sphere function for fitness evaluation.
    The individual is a vector representing some
model parameters.
    The function calculates the sum of squares of
the color, texture, and shape features.
    """
    # Combine the color, texture, and shape
features into a single vector
    features = np.concatenate((color, texture,
shape), axis=1)  # Shape: (num_images, 7)

    # Flatten the features vector and compute the
sum of squares (Sphere function)
    fitness = np.sum(features**2, axis=1).mean()  #
Mean of squared values for all images
```

```
    return fitness

# Initialize population of genetic sequences (each
individual is a sequence of genes)
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

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

# Perform selection: Choose individuals based on
their fitness (roulette wheel selection)
def selection(population, fitness, num_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 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
            # Adjust gene within its specific lower
and upper bounds
            population[i, gene_to_mutate] =
np.random.uniform(lower_bound[gene_to_mutate],
upper_bound[gene_to_mutate])
    return population

# Gene expression: Decode the genetic sequence into
classifier parameters
def gene_expression(individual, fitness_function,
color, texture, shape, labels):
    return fitness_function(individual, color,
texture, shape, labels)
```

```
# 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,
color, texture, shape, labels):
    population =
initialize_population(population_size, num_genes,
lower_bound, upper_bound)
    best_solution = None
    best_fitness = float('inf')

    for generation in range(max_generations):
        fitness = evaluate_fitness(population,
fitness_function, color, texture, shape, labels)

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

        selected_population = selection(population,
fitness, population_size // 2)
        offspring_population =
crossover(selected_population, crossover_rate)
        population = mutation(offspring_population,
mutation_rate, lower_bound, upper_bound)
```

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

    return best_solution, best_fitness

# Simulate a dataset of numerical feature vectors for 50 images, each with color, texture, and shape features
def generate_fake_image_features():
    # Simulate 50 images, each represented by 3 color features, 2 texture features, and 2 shape features
    color = np.random.rand(50, 3)  # 50 images, 3 color features (RGB)
    texture = np.random.rand(50, 2)  # 50 images, 2 texture features (entropy, contrast)
    shape = np.random.rand(50, 2)  # 50 images, 2 shape features (aspect ratio, compactness)
    labels = np.random.randint(0, 2, 50)  # Random binary labels for simplicity
    return color, texture, shape, labels

# Parameters for the algorithm
population_size = 50
num_genes = 1  # Number of genes (we can keep it 1 for simplicity, extending this as needed)
lower_bound = [0.01]  # Lower bound for the individual (C parameter for example)
```

```
upper_bound = [1000]  # Upper bound for the
individual (C parameter for example)
max_generations = 100
mutation_rate = 0.1
crossover_rate = 0.7

# Generate fake image feature data
color, texture, shape, labels =
generate_fake_image_features()

# 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,
sphere_function,
    color, texture, shape, labels)

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

Output:

```
Generation 10/100, Best Fitness: 2.4019580734599
Generation 20/100, Best Fitness: 2.4019580734599
Generation 30/100, Best Fitness: 2.4019580734599
Generation 40/100, Best Fitness: 2.4019580734599
Generation 50/100, Best Fitness: 2.4019580734599
Generation 60/100, Best Fitness: 2.4019580734599
Generation 70/100, Best Fitness: 2.4019580734599
Generation 80/100, Best Fitness: 2.4019580734599
Generation 90/100, Best Fitness: 2.4019580734599
Generation 100/100, Best Fitness: 2.4019580734599

Best Solution Found: [368.67320898]
Best Fitness Value: 2.4019580734599
```