Exercise: Implementing a Simple Genetic Algorithm in Python

Objective

Write a Python program that simulates a genetic algorithm to maximize a function $f(x) = x^2$, where x is represented by a binary string. The goal is for students to apply the five steps of a genetic algorithm: initialization, fitness evaluation, selection, crossover, and mutation.

Instructions

Define the Genetic Algorithm Parameters

- Set population size, mutation rate, crossover rate, and the maximum number of generations.
- Define the length of the binary string to represent x (e.g., 5 bits for values between 0 and 31).

Step 1: Initialization

- Generate a random initial population of binary strings (chromosomes).
- Each binary string represents an individual in the population.

Step 2: Fitness Evaluation

- Write a function to convert each binary string to its decimal representation.
- Define a fitness function, $f(x) = x^2$, and apply it to each individual to get the fitness score.

Step 3: Selection

- Implement a selection process where individuals are chosen based on their fitness scores.
- Use Roulette Wheel Selection or any other selection method you've learned.

Step 4: Crossover

- For each selected pair of individuals, perform crossover at a randomly chosen point to produce offspring.

Step 5: Mutation

- Apply mutation by flipping random bits in the binary strings with a set mutation rate.

Termination

The algorithm should terminate after a certain number of generations or if an optimal solution is found.

Output

- Print the best solution found, including the binary string, its decimal representation, and its fitness score.
- Track the evolution by displaying the maximum fitness per generation.

Python Code Template

Below is a template to guide the students through implementing each step of the genetic algorithm:

```
import random

# Parameters
POP_SIZE = 10       # Population size
GEN_LENGTH = 5      # Length of binary string (for x between 0 and 31)
MUTATION_RATE = 0.1
CROSSOVER_RATE = 0.8
MAX_GENERATIONS = 20

# Fitness function
def fitness(x):
    return x ** 2

# Step 1: Initialize Population
def initialize_population():
    population = []
    for _ in range(POP_SIZE):
        chromosome = ''.join(random.choice('01') for _ in range(GEN_LENGTH))
        population.append(chromosome)
    return population

# Convert binary string to decimal
def binary_to_decimal(binary_str):
    return int(binary_str, 2)

# Step 2: Calculate Fitness for each individual
def calculate_fitness(population):
    return [fitness(binary_to_decimal(individual)) for individual in population]

# Step 3: Selection using Roulette Wheel
def select_parents(population, fitness_scores):
    total_fitness = sum(fitness_scores)
    selection_probs = [f / total_fitness for f in fitness_scores]
    parents = random.choices(population, weights=selection_probs, k=POP_SIZE)
    return parents

# Step 4: Crossover
def crossover(parent1, parent2):
    if random.random() < CROSSOVER_RATE:
        point = random.randint(1, GEN_LENGTH - 1)
        child1 = parent1[:point] + parent2[point:]
```

```python
        child2 = parent2[:point] + parent1[point:]
        return child1, child2
    return parent1, parent2

# Step 5: Mutation
def mutate(chromosome):
    mutated = ''.join(
        '1' if bit == '0' and random.random() < MUTATION_RATE else
        '0' if bit == '1' and random.random() < MUTATION_RATE else bit
        for bit in chromosome
    )
    return mutated

# Main Genetic Algorithm function
def genetic_algorithm():
    population = initialize_population()
    for generation in range(MAX_GENERATIONS):
        fitness_scores = calculate_fitness(population)
        parents = select_parents(population, fitness_scores)

        # Create next generation through crossover and mutation
        next_generation = []
        for i in range(0, POP_SIZE, 2):
            parent1, parent2 = parents[i], parents[i + 1]
            child1, child2 = crossover(parent1, parent2)
            next_generation.append(mutate(child1))
            next_generation.append(mutate(child2))

        population = next_generation
        best_individual = max(population, key=lambda ind: fitness(binary_to_decimal(ind)))
        best_fitness = fitness(binary_to_decimal(best_individual))
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}, Best Individual = {best_individual}")

    # Final result
    best_individual = max(population, key=lambda ind: fitness(binary_to_decimal(ind)))
    best_fitness = fitness(binary_to_decimal(best_individual))
    print("\nBest solution found:")
    print(f"Binary: {best_individual}, Decimal: {binary_to_decimal(best_individual)}, Fitness: {best_fitness}")

# Run the Genetic Algorithm
genetic_algorithm()
```

Tasks for Students

- **Run the Code**: Execute the program and observe how the population evolves over generations.
- **Modify Parameters**: Experiment with different values for `POP_SIZE`, `MUTATION_RATE`, `CROSSOVER_RATE`, and `MAX_GENERATIONS` to see how it affects the results.
- **Add Visualizations**: Plot the fitness values over generations to visualize the algorithm's progress.
- **Extend the Algorithm**: Modify the fitness function to test other optimization problems and observe the genetic algorithm's adaptability.
- **Discuss the Results**: Analyze the results and explain how each parameter influenced the outcome.

Lecture Note by: H.L.N. Himanshi - Department of Artificial Intelligence and Non-Linear Analysis, Faculty of Mathematics and Computer Science, University of Lodz, Poland