GENETIC ALGORITHM APPLICATION-FUNCTION OPTIMIZATION PROBLEM

import random

# Define the function to maximize: f(x) = x^2

def fitness(x):

    return x \*\* 2

# Generate a random individual (chromosome)

def generate\_individual(bits=5):

    return ''.join(random.choice('01') for \_ in range(bits))

# Convert binary string to decimal number

def binary\_to\_decimal(binary\_str):

    return int(binary\_str, 2)

# Create initial population

def create\_population(pop\_size=10, bits=5):

    return [generate\_individual(bits) for \_ in range(pop\_size)]

# Selection: tournament selection

def select\_population(population, fitness\_values):

    selected = []

    for \_ in range(len(population) // 2):

        tournament = random.sample(list(zip(population, fitness\_values)), 2)

        winner = max(tournament, key=lambda x: x[1])

        selected.append(winner[0])

    return selected

# Crossover: single-point crossover

def crossover(parent1, parent2):

    point = random.randint(1, len(parent1) - 1)  # Random crossover point

    child1 = parent1[:point] + parent2[point:]

    child2 = parent2[:point] + parent1[point:]

    return child1, child2

# Mutation: flip a random bit

def mutate(individual, mutation\_rate=0.01):

    individual = list(individual)

    for i in range(len(individual)):

        if random.random() < mutation\_rate:

            individual[i] = '1' if individual[i] == '0' else '0'

    return ''.join(individual)

# Main Genetic Algorithm function

def genetic\_algorithm(pop\_size=10, generations=100, mutation\_rate=0.01, bits=5):

    # Create initial population

    population = create\_population(pop\_size, bits)

    # Ensure the population size is even (if it's odd, remove one individual)

    if len(population) % 2 != 0:

        population = population[:-1]  # Remove one individual to make it even

    for generation in range(generations):

        # Evaluate fitness of each individual

        fitness\_values = [fitness(binary\_to\_decimal(individual)) for individual in population]

        # Check if fitness\_values is empty or has issues

        if not fitness\_values:

            print(f"Error: Fitness values are empty in generation {generation + 1}")

            return None, None

        # Print the best individual in the current generation

        best\_individual = population[fitness\_values.index(max(fitness\_values))]

        best\_fitness = max(fitness\_values)

        print(f"Generation {generation + 1}: Best Individual = {best\_individual}, Fitness = {best\_fitness}")

        # Select the best individuals

        selected\_population = select\_population(population, fitness\_values)

        # Ensure the selected population has an even number of individuals

        if len(selected\_population) % 2 != 0:

            selected\_population = selected\_population[:-1]  # Remove one individual if the list is odd

        # Perform crossover

        next\_generation = []

        for i in range(0, len(selected\_population), 2):

            parent1 = selected\_population[i]

            parent2 = selected\_population[i + 1]

            child1, child2 = crossover(parent1, parent2)

            next\_generation.extend([child1, child2])

        # Perform mutation

        next\_generation = [mutate(individual, mutation\_rate) for individual in next\_generation]

        # Replace population with new generation

        population = next\_generation

    # Final evaluation after all generations

    final\_fitness\_values = [fitness(binary\_to\_decimal(individual)) for individual in population]

    if not final\_fitness\_values:

        print("Error: Final fitness values are empty.")

        return None, None

    best\_individual = population[final\_fitness\_values.index(max(final\_fitness\_values))]

    return best\_individual, max(final\_fitness\_values)

# Run the genetic algorithm

best\_solution, best\_fitness = genetic\_algorithm(pop\_size=20, generations=50, mutation\_rate=0.01, bits=5)

if best\_solution is not None:

    # Decode the best solution

    best\_solution\_decimal = binary\_to\_decimal(best\_solution)

    print(f"Best solution found: {best\_solution\_decimal} with fitness {best\_fitness}")

else:

    print("No solution found due to error in fitness calculation.")

OUTPUT:

