LAB 10 Genetic Algorithm

Genetic Algorithm

A genetic algorithm is a search and optimization technique inspired by the process of natural selection. It mimics the principles of genetics and evolution to solve complex problems. Here's a detailed explanation of the genetic algorithm along with an example code implementation:

Initialization:

Define the size of the population, which consists of a set of individuals.

Each individual represents a potential solution to the problem and is encoded as a string of genes (binary, integer, or real values).

Generate an initial population randomly or using some heuristics.

Fitness Evaluation:

Evaluate the fitness of each individual in the population.

The fitness function measures how well an individual solves the problem.

It assigns a numerical value (fitness score) indicating the quality of the solution.

Selection:

Select individuals from the population based on their fitness scores.

Individuals with higher fitness scores have a higher chance of being selected.

Selection methods include roulette wheel selection, tournament selection, or rank-based selection.

Reproduction:

Generate offspring by combining genetic material from selected individuals.

Common reproduction techniques are crossover and mutation:

Crossover: Randomly select two parents and exchange genetic information to create offspring.

Mutation: Randomly modify genes in an individual to introduce new variations.

Replacement:

Replace some individuals in the population with the newly created offspring.

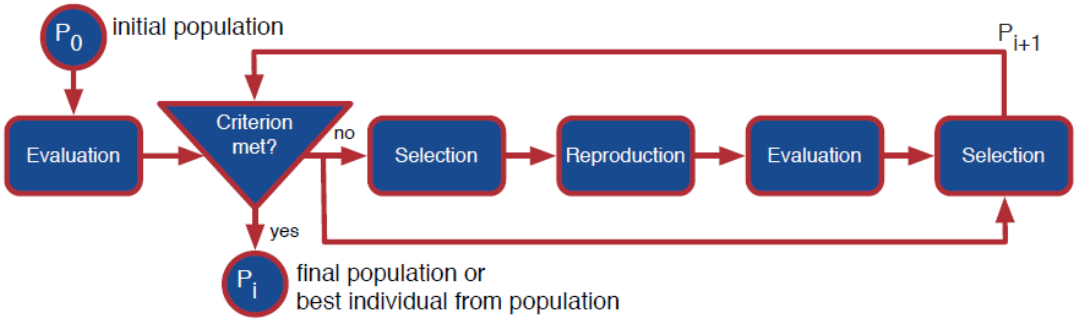
The replacement can be based on the fitness scores of individuals or other criteria.

Termination:

Decide when to stop the algorithm based on a termination condition (e.g., a maximum number of generations or a satisfactory solution).

If the termination condition is not met, go back to step 2 and repeat the process.

Genetic algorithms are usually used to identify optimal solutions to complex problems. This can clearly be easily mapped to search methods, which are aiming toward a similar goal. Genetic algorithms can thus be used to search for solutions to multi-value problems where the closeness of any attempted solution to the actual solution (**fitness**) can be readily evaluated. In short, a **population** of possible solutions (**chromosomes**) is generated, and a fitness value for each chromosome is determined. This fitness is used to determine the likelihood that a given chromosome will survive to the next generation, or reproduce. Reproduction is done by applying **crossover** to two (or more) chromosomes, whereby features (**genes**) of each chromosome are combined together. Mutation is also applied, which involves making random changes to particular genes.



Now, let's see an example implementation of the genetic algorithm in Python for a simple optimization problem of finding the maximum value of a function.

```
import random

# Genetic Algorithm parameters
population_size = 100
chromosome_length = 20
mutation_rate = 0.01
num_generations = 50

# Define the fitness function
def fitness_function(chromosome):
    # Convert binary chromosome to decimal value
    value = int(''.join(map(str, chromosome)), 2)
    # Evaluate the fitness (maximum value of a function)
    fitness = value * value
    return fitness

# Generate an initial population
def generate_population():
    population = []
```

```
    for _ in range(population_size):
        chromosome = [random.randint(0, 1) for _ in
range(chromosome_length)]
        population.append(chromosome)
    return population

# Perform crossover between two parents
def crossover(parent1, parent2):
    crossover_point = random.randint(0, chromosome_length - 1)
    child = parent1[:crossover_point] + parent2[crossover_point:]
    return child

# Perform mutation on an individual
def mutate(individual):
    for i in range(chromosome_length):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip the bit

# Genetic Algorithm main loop
def genetic_algorithm():
    population = generate_population()

    for generation in range(num_generations):
        # Evaluate fitness of each individual
        fitness_scores = [fitness_function(chromosome) for chromosome
in population]

        # Select parents for reproduction
        parents = random.choices(population, weights=fitness_scores,
k=2)

        # Create offspring through crossover
        offspring = [crossover(parents[0], parents[1]) for _ in
range(population_size)]

        # Apply mutation to the offspring
        for individual in offspring:
            mutate(individual)

        # Replace the old population with the offspring
        population = offspring

    # Find the best individual (maximum fitness)
    best_individual = max(population, key=fitness_function)
    best_fitness = fitness_function(best_individual)

    return best_individual, best_fitness
```

```
# Run the genetic algorithm
best_solution, best_fitness = genetic_algorithm()

# Print the result
print("Best Solution:", best_solution)
print("Best Fitness:", best_fitness)
```

Here's a step-by-step description of the provided code:

Import the necessary modules:

```
import random
```

Define the parameters for the genetic algorithm:

```
population_size = 100  # Size of the population
chromosome_length = 20  # Length of each chromosome
mutation_rate = 0.01  # Probability of mutation
num_generations = 50  # Number of generations to run
```

Define the fitness function:

```
def fitness_function(chromosome):
    value = int(''.join(map(str, chromosome)), 2)  # Convert binary
chromosome to decimal value
    fitness = value * value  # Evaluate the fitness (maximum value of a
function)
    return fitness
```

The fitness function takes a chromosome (individual) as input, converts it from binary to decimal representation, evaluates its fitness (maximum value of a function), and returns the fitness score.

Generate an initial population:

```
def generate_population():
    population = []
    for _ in range(population_size):
        chromosome = [random.randint(0, 1) for _ in
range(chromosome_length)]  # Create a random binary chromosome
        population.append(chromosome)
```

```
    return population
```

The generate_population function creates a list of individuals (population) by generating random binary chromosomes of a specified length.

Perform crossover between two parents:

```
def crossover(parent1, parent2):
    crossover_point = random.randint(0, chromosome_length - 1)  # 
Select a random crossover point
    child = parent1[:crossover_point] + parent2[crossover_point:]  # 
Combine genetic information of parents to create a child
    return child
```

The crossover function takes two parent chromosomes, selects a random crossover point, and combines their genetic information to create a child chromosome.

Perform mutation on an individual:

```
def mutate(individual):
    for i in range(chromosome_length):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual[i]  # Flip the bit 
(mutation)
```

The mutate function iterates through each gene of an individual's chromosome and with a certain probability (mutation_rate), it flips the bit (mutation) to introduce variation.

Implement the main loop of the genetic algorithm:

```
def genetic_algorithm():
    population = generate_population()  # Generate the initial 
population

    for generation in range(num_generations):
        fitness_scores = [fitness_function(chromosome) for chromosome 
in population]  # Evaluate fitness of each individual

        parents = random.choices(population, weights=fitness_scores, 
k=2)  # Select parents for reproduction

        offspring = [crossover(parents[0], parents[1]) for _ in 
range(population_size)]  # Create offspring through crossover
```

```
        for individual in offspring:
            mutate(individual)  # Apply mutation to the offspring

        population = offspring  # Replace the old population with the
offspring

    best_individual = max(population, key=fitness_function)  # Find the
best individual (maximum fitness)
    best_fitness = fitness_function(best_individual)  # Get the fitness
score of the best individual

    return best_individual, best_fitness
```

The genetic_algorithm function implements the main loop of the genetic algorithm. It iterates through a specified number of generations. In each generation, it evaluates the fitness of each individual, selects parents for reproduction based on their fitness scores, creates offspring through crossover, applies mutation to the offspring, and replaces the old population with the offspring.

Example to solve the Task 1 related problem by using genetic algorithm:

```
import random

# Define the fitness function
def fitness_function(x):
    return (-x**2)/10 + 3*x

# Function to generate a random chromosome
def generate_chromosome():
    return [random.randint(0, 1) for _ in range(5)]

# Function to decode chromosome to x value
def decode_chromosome(chromosome):
    x = 0
    for bit in chromosome:
        x = (x << 1) | bit
    return x

# Function to perform crossover between two chromosomes
def crossover(chromosome1, chromosome2):
    crossover_point = random.randint(1, len(chromosome1) - 1)
    new_chromosome1 = chromosome1[:crossover_point] +
chromosome2[crossover_point:]
    new_chromosome2 = chromosome2[:crossover_point] +
chromosome1[crossover_point:]
    return new_chromosome1, new_chromosome2
```

```
# Function to perform mutation on a chromosome
def mutate(chromosome):
    mutation_point = random.randint(0, len(chromosome) - 1)
    chromosome[mutation_point] = 1 - chromosome[mutation_point]  # Flip
the bit
    return chromosome

# Initialize the population
population_size = 10
population = [generate_chromosome() for _ in range(population_size)]

# Main loop
generation = 1
while True:
    # Calculate fitness values for each chromosome
    fitness_values = [fitness_function(decode_chromosome(chromosome))
for chromosome in population]

    # Check termination condition
    best_fitness = max(fitness_values)
    if best_fitness >= 0.9 * max(fitness_values):
        break

    # Selection - Roulette wheel selection
    selection_probabilities = [fitness / sum(fitness_values) for
fitness in fitness_values]
    selected_indices = random.choices(range(population_size),
weights=selection_probabilities, k=population_size)

    # Create the new population through crossover and mutation
    new_population = []
    for i in range(0, population_size, 2):
        chromosome1 = population[selected_indices[i]]
        chromosome2 = population[selected_indices[i+1]]
        chromosome1, chromosome2 = crossover(chromosome1, chromosome2)
        if generation % 3 == 0:  # Apply mutation every 3 generations
            chromosome1 = mutate(chromosome1)
            chromosome2 = mutate(chromosome2)
        new_population.extend([chromosome1, chromosome2])

    population = new_population
    generation += 1

# Print the result
best_chromosome = population[fitness_values.index(max(fitness_values))]
best_x = decode_chromosome(best_chromosome)
best_fitness = fitness_function(best_x)
print("Best Chromosome:", best_chromosome)
```

```
print("Best x:", best_x)
print("Best Fitness:", best_fitness)
```

In this implementation:

- The fitness function is defined as fitness_function(x).
- The generate_chromosome() function generates a random chromosome of length 5.
- The decode_chromosome(chromosome) function decodes a chromosome to its corresponding x value.
- The crossover(chromosome1, chromosome2) function performs crossover between two chromosomes.
- The mutate(chromosome) function performs mutation on a chromosome by flipping a random bit.
- The main loop continues until one of the candidate's fitness function value is greater or equal to 90%.
- Within each generation, fitness values are calculated for each chromosome, and the termination condition is checked.
- Roulette wheel selection is used to select chromosomes for the next generation.
- Crossover and mutation are applied to create the new population.
- The best chromosome, its corresponding x value, and fitness are printed as the result.