Lab 1

Genetic Algorithm for Optimization Problems

29/8/2025

LAB - II

Genetic Algorithm

5 main phases: Initializ^n, Fitness Assignment, selec^n, Crossover, Terminat^n

Steps:

1) selecting encoding techniques

0 to 31

2) select initial population

String No.	4 Initial popul^n	x value	Fitness $f(x) = x^2$	Prob $f(x)/\Sigma f(x)$	% prob	Expected count $f(x)/(avg \Sigma f(x))$	Actual count
1	01100	12	144	0.1247	12.47	0.49	1
2	11001	25	625	0.5411	54.11	2.164	2
3	00101	5	25	0.0216	2.16	0.086	0
4	10011	19	361	0.3125	31.25	1.25	1
			1155/4=288.75				

3) Select mating ~~prob~~ pool

String No.	Mating pool	Crossover point	Offspring after crossover	x value	fitness $f(x) = x^2$
1	01100	4	01101	13	169
2	11001		11000	24	576
3	11001	2	11011	27	729
4	10011		10001	17	289

4) Crossover: Random 4 + 2

Max value 729

5) Mutation

String No.	Offspring after crossover	Mutat^n chromosome for offspring	Offspring after mutat^n	x value	fitness $f(x)=x^2$
1	01101	10000	11101	29	841
2	11000	00000	11000	24	576
3	11011	00000	11011	27	729
4	10001	00101	10100	20	400
					2546/4=636

```
import random
def fitness(x):
    return x**2

POPULATION_SIZE=4
CHROMOSOME_LENGTH=5
MUTATION_RATE=0.1
GENERATIONS=10
def binary-to-decimal(binary):
    return int(binary,2)
def decimal-to-binary(n):
    return format(n,f'0{CHROMOSOME_LENGTH}b')
def initialize-population():
    return [decimal-to-binary(random.randint(0, 2**CHROMOSOME_LENGTH-1)) for _ in range(POPULATION_SIZE)]
def evaluate-population(population):
    return [fitness(binary-to-decimal(individual)) for individual in population]
def select-parents(population, fitnesses):
    parents=[]
    for _ in range(2):
    i,j =random.sample(range(len(population)), 2)
    if fitnesses[i]>fitnesses[j]:
        parents.append(population[i])
    else:
        parents.append(population[j])
    return parents
def crossover(parent1, parent2):
    point=random.randint(1, CHROMOSOME_LENGTH-1)
    Child1 = parent1[:point] + parent2[point:]
    Child2 = parent2[:point] + parent1[point:]
    return Child1, child2
```

```
def mutate(individual):
    mutated = ''
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
        else:
            mutated += bit
    return mutated

def genetic_algorithm():
    population = initialize_population()
    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)
        new_population = []
        while len(new_population) < POPULATION SIZE:
            parent1, parent2 = select_parents(population, fitnesses)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1)
            child2 = mutate(child2)
            new_population.extend([child1, child2])
        population = new_population[:POPULATION_SIZE]
        best = max(population, key=lambda x: fitness(binary_to_decimal
                                                                  (x)))
        print(f"Generation {generation+1}: Best = {binary_to_decimal(best)},
                              : Fitness = {fitness(binary_to_decimal
                                                                  (best))}")
    best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
    print("Best solution:", binary_to_decimal(best))
    print("Fitness:", fitness(binary_to_decimal(best)))

genetic_algorithm()
```

Code:

```
import random

# 1. Define problem (fitness function)
def fitness(x):
    return x ** 2

# 2. Initialize parameters
POPULATION_SIZE = 4
CHROMOSOME_LENGTH = 5  # We'll use 5-bit binary strings, i.e. values 0–31
MUTATION_RATE = 0.1
GENERATIONS = 10

# Helper: Binary to decimal
def binary_to_decimal(binary):
    return int(binary, 2)

# Helper: Decimal to binary
def decimal_to_binary(n):
    return format(n, f'0{CHROMOSOME_LENGTH}b')

# 3. Create initial population
def initialize_population():
    return [decimal_to_binary(random.randint(0, 2**CHROMOSOME_LENGTH - 1)) for _ in
range(POPULATION_SIZE)]

# 4. Evaluate fitness of the population
```

```
def evaluate_population(population):
    return [fitness(binary_to_decimal(individual)) for individual in population]

# 5. Selection (Tournament Selection)
def select_parents(population, fitnesses):
    parents = []
    for _ in range(2):
        i, j = random.sample(range(len(population)), 2)
        if fitnesses[i] > fitnesses[j]:
            parents.append(population[i])
        else:
            parents.append(population[j])
    return parents

# 6. Crossover (Single-point)
def crossover(parent1, parent2):
    point = random.randint(1, CHROMOSOME_LENGTH - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

# 7. Mutation (Bit-flip)
def mutate(individual):
    mutated = ''
    for bit in individual:
        if random.random() < MUTATION_RATE:
            mutated += '1' if bit == '0' else '0'
```

```
        else:
            mutated += bit
    return mutated

# 8. Iteration
def genetic_algorithm():
    population = initialize_population()

    for generation in range(GENERATIONS):
        fitnesses = evaluate_population(population)

        new_population = []

        while len(new_population) < POPULATION_SIZE:
            # Select
            parent1, parent2 = select_parents(population, fitnesses)
            # Crossover
            child1, child2 = crossover(parent1, parent2)
            # Mutation
            child1 = mutate(child1)
            child2 = mutate(child2)
            # Add to new population
            new_population.extend([child1, child2])

        # Replace old population with new (trim if needed)
        population = new_population[:POPULATION_SIZE]
```

```
        # Debug info
        best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
        print(f"Generation {generation+1}: Best = {binary_to_decimal(best)}, Fitness = {fitness(binary_to_decimal(best))}")

    # 9. Output best solution
    best = max(population, key=lambda x: fitness(binary_to_decimal(x)))
    print("\nBest solution:", binary_to_decimal(best))
    print("Fitness:", fitness(binary_to_decimal(best)))

# Run the genetic algorithm
genetic_algorithm()
```

Output:

```
Shreya Raj 1BM23CS317
Generation 1: Best = 28, Fitness = 784
Generation 2: Best = 28, Fitness = 784
Generation 3: Best = 30, Fitness = 900
Generation 4: Best = 30, Fitness = 900
Generation 5: Best = 31, Fitness = 961
Generation 6: Best = 31, Fitness = 961
Generation 7: Best = 31, Fitness = 961
Generation 8: Best = 31, Fitness = 961
Generation 9: Best = 31, Fitness = 961
Generation 10: Best = 31, Fitness = 961

Best solution: 31
Fitness: 961
```