LAB 01 **(09/10/24)**

Genetic Algorithm for Optimization Problems:
Genetic Algorithms (GA) are inspired by the process of natural selection and genetics, where the fittest individuals are selected for reproduction to produce the next generation. GAs are widely used for solving optimization and search problems. Implement a Genetic Algorithm using Python to solve a basic optimization problem, such as finding the maximum value of a mathematical function.

IMPLEMENTATION:
1. Define the Problem: Create a mathematical function to optimize.
2. Initialize Parameters: Set the population size, mutation rate, crossover rate, and number of generations.
3. Create Initial Population: Generate an initial population of potential solutions.
4. Evaluate Fitness: Evaluate the fitness of each individual in the population.
5. Selection: Select individuals based on their fitness to reproduce.
6. Crossover: Perform crossover between selected individuals to produce offspring.
7. Mutation: Apply mutation to the offspring to maintain genetic diversity.
8. Iteration: Repeat the evaluation, selection, crossover, and mutation processes for a fixed number of generations or until convergence criteria are met.
9. Output the Best Solution: Track and output the best solution found during the generations

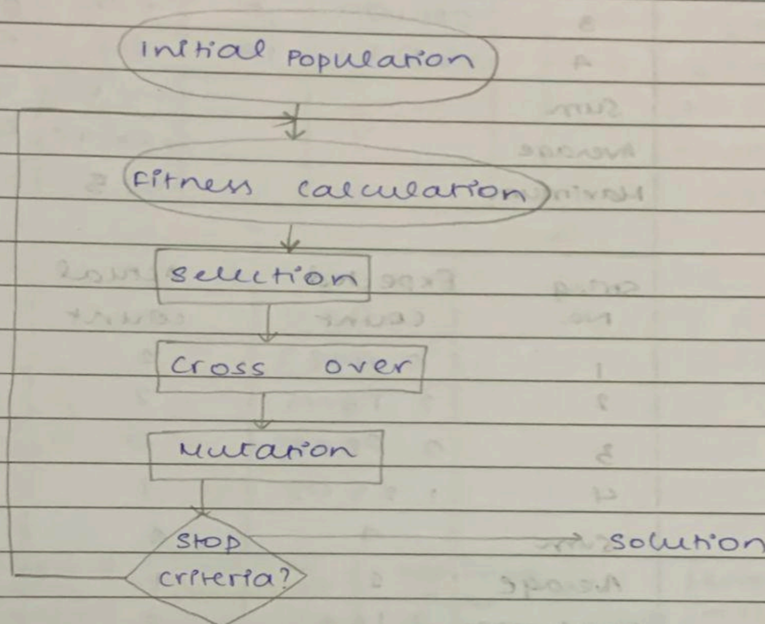
ALGORITHM/LOGIC -

GENETIC ALGORITHM

Date 09/10/24
Page 01

AIM: High quality solution for optimization Problem.

FLOWCHART:



solved Example →

$f(x) = x^2$.

$0 < x < 31$ i.e, 0(00000) < x < 31(11111)

Initial population = 4 (randomly selected)

Date 09/10/24
Page 02

Fitness Calculation

String No.	Initial population (Randomly selected)	X Value	Fitness $f(x) = x^2$	Prob	Prob (%) (Prob × 100)
1	01100	12	144	0.1247	12.47
2	11001	25	625	0.5411	54.11
3	00101	5	25	0.0216	2.16
4	10011	19	181	0.3126	31.26
Sum			1155	1.0	100
Average			288.75	0.25	25
Maximum			625	0.5411	54.11

String No.	Expected count	Actual count
1	0.4987	1
2	2.1645	2
3	0.0866	0
4	1.2502	1
Sum	4	4
Average	1	1
Maximum	2.1645	2

1. Probability $= \dfrac{f(x)}{\Sigma f(x)}$

2. Expected count $= \dfrac{f(x_i)}{Avg(\Sigma f(x))}$

3. Actual count = Rounded value of expected count.

selection + crossover

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
Sum					1763
Average					440.75
Maximum					729

- 0110|0 ⇒ 01101
  1100|1 ⇒ 11000

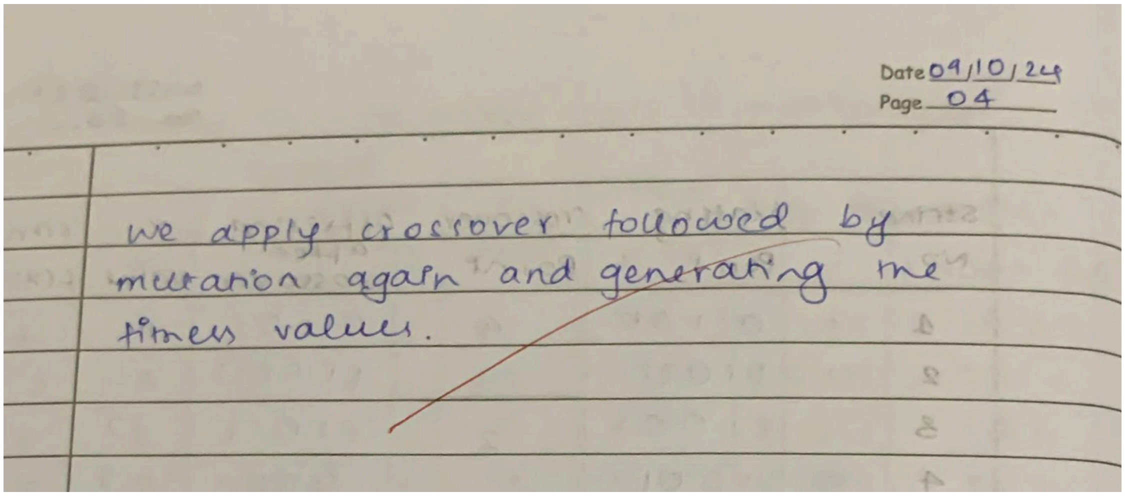
- 11|001 ⇒ 11011
  10|011 ⇒ 10001

Mutation

String No.	Offspring after crossover	Mutation chromosome for flipping	Offspring after mutation	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
Sum					2546
Average					636.5
Maximum					841

- We repeat the same process until we get the maximum possible value for the function $f(x) = x^2$.

For this we select some solution from the offspring after mutation and then


Date 04/10/24
Page 04

we apply crossover followed by mutation again and generating the fitness values.

INPUT -

```
import random
import numpy as np

# Define the problem: the function to optimize
def fitness_function(x):
    return x**2

# Initialize parameters
population_size = 10
mutation_rate = 0.1
crossover_rate = 0.7
num_generations = 50
search_space = (-10, 10)
def initialize_population(size, bounds):
    return [random.uniform(bounds[0], bounds[1]) for _ in range(size)]

def evaluate_population(population):
    return [fitness_function(ind) for ind in population]

def select_parents(population, fitness):
    total_fitness = sum(fitness)
    selection_probs = [f / total_fitness for f in fitness]
    return np.random.choice(population, size=2, p=selection_probs, replace=False)

def crossover(parent1, parent2):
    if random.random() < crossover_rate:
        alpha = random.random()
        child1 = alpha * parent1 + (1 - alpha) * parent2
```

```
        child2 = alpha * parent2 + (1 - alpha) * parent1
        return child1, child2
    return parent1, parent2

def mutate(individual, bounds):
    if random.random() < mutation_rate:
        mutation_value = random.uniform(-1, 1)          individual = individual + mutation_value
        individual = max(min(individual, bounds[1]), bounds[0])
    return individual

# Genetic Algorithm
def genetic_algorithm():
    population = initialize_population(population_size, search_space)

    best_individual = None
    best_fitness = -float('inf')
    for generation in range(num_generations):
        fitness = evaluate_population(population)
        max_fitness_idx = np.argmax(fitness)
        if fitness[max_fitness_idx] > best_fitness:
            best_fitness = fitness[max_fitness_idx]
            best_individual = population[max_fitness_idx]

        new_population = []
        while len(new_population) < population_size:
            parent1, parent2 = select_parents(population, fitness)
            child1, child2 = crossover(parent1, parent2)
            child1 = mutate(child1, search_space)
            child2 = mutate(child2, search_space)
            new_population.extend([child1, child2])
        population = new_population[:population_size]
        print(f"Generation {generation + 1}: Best Fitness = {best_fitness}")
    print("Best solution:", best_individual)
    print("Best fitness:", best_fitness)

# Run the Genetic Algorithm
genetic_algorithm()
```

OUTPUT-

```
Generation 1: Best Fitness = 77.56328648666843
Generation 2: Best Fitness = 77.56328648666843
Generation 3: Best Fitness = 77.56328648666843
Generation 4: Best Fitness = 77.56328648666843
Generation 5: Best Fitness = 77.56328648666843
Generation 6: Best Fitness = 80.16980384513786
Generation 7: Best Fitness = 80.16980384513786
Generation 8: Best Fitness = 80.16980384513786
Generation 9: Best Fitness = 80.16980384513786
Generation 10: Best Fitness = 80.16980384513786
Generation 11: Best Fitness = 80.16980384513786
Generation 12: Best Fitness = 80.16980384513786
Generation 13: Best Fitness = 80.16980384513786
Generation 14: Best Fitness = 80.16980384513786
Generation 15: Best Fitness = 80.16980384513786
Generation 16: Best Fitness = 80.16980384513786
Generation 17: Best Fitness = 80.16980384513786
Generation 18: Best Fitness = 80.16980384513786
Generation 19: Best Fitness = 80.16980384513786
Generation 20: Best Fitness = 80.16980384513786
Generation 21: Best Fitness = 80.16980384513786
Generation 22: Best Fitness = 80.16980384513786
Generation 23: Best Fitness = 80.16980384513786
Generation 24: Best Fitness = 80.16980384513786
Generation 25: Best Fitness = 80.16980384513786
Generation 26: Best Fitness = 80.16980384513786
Generation 27: Best Fitness = 80.16980384513786
Generation 28: Best Fitness = 80.16980384513786
Generation 29: Best Fitness = 80.16980384513786
Generation 30: Best Fitness = 80.16980384513786
Generation 31: Best Fitness = 80.16980384513786
Generation 32: Best Fitness = 80.16980384513786
Generation 33: Best Fitness = 80.16980384513786
Generation 34: Best Fitness = 80.16980384513786
Generation 35: Best Fitness = 80.16980384513786
Generation 36: Best Fitness = 80.16980384513786
Generation 37: Best Fitness = 80.16980384513786
Generation 38: Best Fitness = 80.16980384513786
Generation 39: Best Fitness = 80.16980384513786
Generation 40: Best Fitness = 80.16980384513786
Generation 41: Best Fitness = 80.16980384513786
Generation 42: Best Fitness = 80.16980384513786
Generation 43: Best Fitness = 80.16980384513786
Generation 44: Best Fitness = 80.16980384513786
Generation 45: Best Fitness = 80.16980384513786
Generation 46: Best Fitness = 80.16980384513786
```

```
Generation 45: Best Fitness = 80.16980384513786
Generation 46: Best Fitness = 80.16980384513786
Generation 47: Best Fitness = 80.16980384513786
Generation 48: Best Fitness = 80.16980384513786
Generation 49: Best Fitness = 80.16980384513786
Generation 50: Best Fitness = 80.16980384513786
Best solution: 8.953759201873694
Best fitness: 80.16980384513786
```