Lab-2

Genetic Algorithm

$F(x) = x^2$

1) Initialize Parameters:

i) Population Size: Choose a population size, say 6 individuals.

ii) Binary Representation: Each individual will be a 5-bit binary string.

iii) Randomly Initialize Population: Generate to random binary strings.

Population: ["10101", "00111", "01100", "11001", "10010", "00010"]

Fitness Evaluation:

Convert binary to decimal

$10101 \rightarrow 21 \rightarrow 21^2 = 441$

$00111 \rightarrow 7 \rightarrow 7^2 = 49$

$01100 \rightarrow 12 \rightarrow 12^2 = 144$

$11001 \rightarrow 25 \rightarrow 25^2 = 625$

$10010 \rightarrow 18 \rightarrow 18^2 = 324$

$00010 \rightarrow 2 \rightarrow 2^2 = 4$

• Selection

use ~~of~~ a method like tournament.

on selection: ["11001", "10101", "10010"]

• Crossover

Pair 1: 11001 and 10101 →
on crossover
→ offspring: 11001, 10101.

Pair 2: 10010 and a ~~parent~~ random parent
10010 and 00111.
offspring: 10011, 00110.

• Offspring after Crossover

:- New offspring: ["11001", "10101", "10011", "00110"]

• Mutation
If we mutate 10011 → 10001

• Replacement
Replace the old population with the new offspring (or keep the best individuals).
• New Population: ["11001", "10100", "00100"]

· Iteration
Repeat steps 2-6 for a predetermined number of generations or until convergence

```
import random
import numpy as np

def fitness_function(x):
    return x**2

population_size = 10
mutation_rate = 0.01
crossover_rate = 0.8
num_generations = 100
gene_length = 10

def create_population(size, gene_length):
    return [np.random.randint(0, 2, gene_length).
    tolist() for _ in range(size)]

def binary_to_decimal(binary):
    binary_str = "".join(str(bit) for bit in
    binary)
    return int(binary_str, 2) / (2 ** gene_length - 1) * 20 - 6

def evaluate_populations(population):
    return [fitness_function(binary_to_decimal(individual))
     for individual in population]

def select(population, fitness_score):
        total_fitness = sum(fitness_scores)
```

```
selection_probs = [fitness / total_fitness for fitness in
  fitness_scores]
return population [np.random.choice(range(len(population)),
  p= selection_probs)]

def mutate (individual):
    for i in range (gene_length):
        if random.random() < mutation_rate:
            individual[i] = 1 - individual [i]
    return individual.

def genetic_algorithm():
    population = create_population (population_size, gene_
              length)
    for generation in range (num_generations):
        fitness scores = evaluate_population (population)

        best_fitness = max (fitness_score)
        best_individual = population [fitness_scores_
          index (best_fitness)]

        print (f"Generation {generation}: Best Fitness: =
              {best_fitness, 4f}")

        new_population = []
        while len (new_population) < population size:
        parent1 = select (population, fitness_scores)
        parent2 = "
        offspring = crossover (parent1, parent2)
        new_population.extend ([mutate (child) for child in
         offspring])
```

Date
Page

```
population = new_population [: population_size]
best_fitness = max(fitness_scores)
best_individual = population [fitness_scores. index (best_
fitness)])
best_solution = binary_to_decimal (best_individual)
```