Date 18/11/24
Page 11

LAB 2

Genetic Algorithm

$F(x) = x^2$

Step 1 : Initialization

1 Population size: choose a population size, say 6 individuals

2 Binary Representation: Each individual will be a 5-bit binary string

3 Randomly Initialize Population: Generate 6 random binary strings.

Step 2 : Fitness Evaluation

convert binary to decimal

10101 = 441

00111 = 49

01100 = 144

11001 = 625

10010 = 324

00010 = 4

Fitness Values : [441, 49, 144, 625, 324, 4

Step 3 : Selection

Use a method like tournament selection, we'll use tournament selection

selected individuals : ["11001", "10101", "10010"]

Step 4 : Crossover

Pair selected individuals to create offspring (single-point crossover)

Pair 1: 11001 & 10101 → crossover at point 2
→ offspring : 11001, 10101.

Pair 2: 10010 & a random parent → 10010 & 00111
→ crossover → offspring : 10011

offspring after crossover

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

Step 5 : Mutation

Introduce mutations to some of the offspring, flip a random bit with a small probability

- If we mutate 10011 → 10001

Date __/__/__
Page 13

Step 6: Replacement

Replace the old population with the new offspring

New population ["11001", "10101", "10001", "00110"]

Step 7: Iteration

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

Code

```
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)]
```

Date / /
Page 19

```
def binary_to_decimal(binary):
    binary_str = ''.join(str(bit) for bit in binary)
    return int(binary_str, 2) / (2**gene_length - 1) * 20 - 10

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

def select(population, fitness_scores):
    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 crossover(parent1, parent2):
    if random() < crossover_rate:
        child1 = parent1[:crossover_point] + parent2[crossover_point:]
        child2 = parent2[:crossover_point] + parent1[crossover_point:]
        return [child1, child2]

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_score = evaluate_population(population)
    best_fitness = max(fitness_score
print(f"Generation {generation}: Best Fitness
        = {best_fitness:.4f}")

.
    new_population = []
    population = new_population[:population
                                -size]

    best_fitness = max(fitness_scores)
    best_individual = population[fitness_scores.
                      index(best_fitness)]
best_solution = binary_to_decimal(best_individu
                                  al)

print(f"\n Best solution found: x = {best_solution
    :.4f}, f(x) = {fitness_function
    (best_solution):.4f}")

genetic_algorithm()
```

output

```
Gen 0 ; BF = 78.2611
Gen 1 ; BF = 80.3503
Gen 2 ; BF = 90.8360
Gen 3 ; BF = 98.8304
Gen 4 ; BF = 81.0508
```

Date
Page 16

Gen 5 : BF = 82.1123
Gen 6 : BF = 82.1123
Gen 7 : BF = 82.1123
Gen 8 : BF = 83.1258
Gen 9 : BF = 85.1273
Gen 10 : BF = 88.1371

Best solution found: x = 8.8856, f(x) = 78.954