LAB INTERNALS- 19/12/2024

1)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.

CODE:

```
#given code
import numpy as np

def objective_function(x):
    return x ** 2
```

```
population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7
value_range = (-10, 10)
```

```
def initialize_population(size, value_range):
    return np.random.uniform(value_range[0], value_range[1], size)
```

```
def evaluate_fitness(population):
    return np.array([objective_function(x) for x in population])
```

```
def selection(population, fitness):
    probabilities = fitness / fitness.sum()
    return population[np.random.choice(len(population), size=2, p=probabilities)]
```

```
def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        return (parent1 + parent2) / 2
    return parent1 if np.random.rand() < 0.5 else parent2
```

```
def mutate(individual, mutation_rate, value_range):
    if np.random.rand() < mutation_rate:
        return np.random.uniform(value_range[0], value_range[1])
    return individual
```

```
def genetic_algorithm():
```

```
    population = initialize_population(population_size, value_range)
    best_solution = None
    best_fitness = -np.inf
```

```
    for generation in range(num_generations):
```

```
        fitness = evaluate_fitness(population)


        current_best_index = np.argmax(fitness)
        if fitness[current_best_index] > best_fitness:
            best_fitness = fitness[current_best_index]
            best_solution = population[current_best_index]


        new_population = []
        for _ in range(population_size):
            parent1, parent2 = selection(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring, mutation_rate, value_range)
            new_population.append(offspring)

        population = np.array(new_population)

    return best_solution, best_fitness


best_solution, best_fitness = genetic_algorithm()

print(f"Best solution found: x = {best_solution:.2f}")
print(f"Maximum value of f(x) = x^2: f(x) = {best_fitness:.2f}")
```

OUTPUT:

```
Best solution found: x = 10.00
Maximum value of f(x) = x^2: f(x) = 99.94
```

Application:

Schedule jobs on given number of machines to minimize total time taken to compute all the jobs, given different machines have different speeds.

CODE:

```
import numpy as np

# Problem-specific parameters
num_jobs = 10
num_machines = 3
job_times = np.random.randint(1, 10, size=num_jobs)  # Random job durations
population_size = 100
num_generations = 50
mutation_rate = 0.1
crossover_rate = 0.7

# Initialize population: Each individual is a list of machine assignments for each job
def initialize_population(size, num_jobs, num_machines):
    return [np.random.randint(0, num_machines, size=num_jobs) for _ in range(size)]
```

```python
# Objective function: Calculate makespan
def makespan(schedule):
    machine_times = np.zeros(num_machines)
    for job, machine in enumerate(schedule):
        machine_times[machine] += job_times[job]
    return machine_times.max()

# Evaluate fitness: Lower makespan is better, so use inverse
def evaluate_fitness(population):
    return np.array([1 / (1 + makespan(individual)) for individual in population])

# Selection: Roulette wheel selection based on fitness
def selection(population, fitness):
    probabilities = fitness / fitness.sum()
    indices = np.random.choice(len(population), size=2, p=probabilities)
    return population[indices[0]], population[indices[1]]

# Crossover: Average the assignments of two parents (analogous to the original style)
def crossover(parent1, parent2):
    if np.random.rand() < crossover_rate:
        return np.array([(p1 + p2) // 2 for p1, p2 in zip(parent1, parent2)], dtype=int)
    return parent1 if np.random.rand() < 0.5 else parent2

# Mutation: Randomly reassign a job to a machine
def mutate(individual, mutation_rate, num_machines):
    if np.random.rand() < mutation_rate:
        job_to_mutate = np.random.randint(0, len(individual))
        individual[job_to_mutate] = np.random.randint(0, num_machines)
    return individual

# Genetic Algorithm
def genetic_algorithm():
    population = initialize_population(population_size, num_jobs, num_machines)
    best_solution = None
    best_fitness = -np.inf

    for generation in range(num_generations):
        fitness = evaluate_fitness(population)
        current_best_index = np.argmax(fitness)
        if fitness[current_best_index] > best_fitness:
            best_fitness = fitness[current_best_index]
            best_solution = population[current_best_index]

        new_population = []
        for _ in range(population_size):
            parent1, parent2 = selection(population, fitness)
            offspring = crossover(parent1, parent2)
            offspring = mutate(offspring, mutation_rate, num_machines)
            new_population.append(offspring)

        population = new_population

    return best_solution, 1 / best_fitness - 1  # Invert fitness to get makespan

# Run the algorithm
best_schedule, min_makespan = genetic_algorithm()

print(f"Job times: {job_times}")
print(f"Best schedule: {best_schedule}")
print(f"Minimum makespan: {min_makespan:.2f}")
```

OUTPUT:

```
Job times: [8 7 1 2 8 4 8 7 5 3]
Best schedule: [0 0 2 0 1 2 2 1 2 1]
Minimum makespan: 18.00
```

Shreya S Rudagi
1BM22CS267