DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJ19ITL504 **DATE: 15-10-24**

COURSE NAME: Artificial Intelligence Laboratory **CLASS: TY-IT**

NAME: Anish Sharma **ROLL No: I011**

EXPERIMENT NO.06

CO/LO: Apply various AI approaches to knowledge intensive problem solving, reasoning, planning and uncertainty.

AIM / OBJECTIVE: Implement Genetic Algorithm to solve an optimization problem in AI.

DESCRIPTION OF EXPERIMENT:

Genetic Algorithms (GAs) are a class of optimization algorithms inspired by the principles of natural selection and genetics. They are part of the evolutionary computing family and are used to find approximate solutions to optimization and search problems. Here's a detailed overview of the theory underlying Genetic Algorithms:

1. Basic Concepts

1.1 Evolutionary Inspiration: GAs are inspired by Charles Darwin’s theory of natural evolution, where the principle of survival of the fittest is applied to solve optimization problems. The process mimics biological evolution, including the concepts of selection, crossover (recombination), and mutation.

1.2 Chromosomes and Genes: In GAs, potential solutions to the problem are represented as chromosomes. Each chromosome consists of genes, which encode the solution's parameters. For example, in a binary GA, genes might be represented as bits (0s and 1s).

2. Key Components of Genetic Algorithms

2.1 Initialization: The algorithm begins by generating an initial population of chromosomes randomly. Each chromosome represents a potential solution to the optimization problem. The size of the population can significantly impact the algorithm's performance.

2.2 Selection: Selection is the process of choosing individuals from the population to create offspring for the next generation. The aim is to give preference to better-performing individuals based on a fitness function. Common selection methods include:

- Roulette Wheel Selection: Individuals are selected based on their fitness proportionate to the total fitness of the population.
- Tournament Selection: A subset of individuals is chosen randomly, and the best

among them is selected for reproduction.

2.3 Crossover (Recombination): Crossover combines parts of two parent chromosomes to produce offspring. This process mimics biological reproduction and aims to exploit the genetic material of both parents. Common crossover techniques include:

- One-Point Crossover: A single crossover point is chosen, and the parts of the chromosomes beyond this point are swapped between the parents.
- Two-Point Crossover: Two crossover points are chosen, and the segments between these points are exchanged between parents.
- Uniform Crossover: Each gene is independently chosen from one of the parents.

2.4 Mutation: Mutation introduces random changes to individual chromosomes to maintain genetic diversity and avoid premature convergence. This process mimics biological mutation and helps explore new areas of the solution space. Common mutation techniques include:

- Bit Flip Mutation: For binary representations, a bit is flipped from 0 to 1 or from 1 to 0.
- Gaussian Mutation: For real-valued representations, a gene value is altered by adding a small Gaussian-distributed random value.

2.5 Fitness Function: The fitness function evaluates how well a chromosome solves the optimization problem. It assigns a fitness score based on how close the chromosome's solution is to the optimal solution. The design of the fitness function is crucial as it directly impacts the performance of the GA.

2.6 Replacement: After creating offspring, the next step is to form the new generation. This can be done through various replacement strategies, such as:

- Generational Replacement: The entire population is replaced by the new offspring.
- Steady-State Replacement: Only a few individuals are replaced at a time, maintaining some of the old population.

2.7 Termination: The GA process continues for a specified number of generations or until a convergence criterion is met. Common termination conditions include:

- Maximum Number of Generations: The algorithm stops after a predefined number of generations.
- Convergence: The algorithm stops when there is little to no improvement in the best fitness value over a set number of generations.

1. Students should select an appropriate problem.

   Choose an optimization problem suitable for the application of Genetic Algorithms.

   - Examples include:
     - The Traveling Salesman Problem (TSP)
     - Knapsack Problem
     - Function Optimization (e.g., minimizing a complex mathematical function)
     - Job Scheduling Problem

Define the Problem:

1. Formulate the problem as an optimization task. Clearly describe the objective function and constraints.
2. Demonstrate local search algorithm.
3. Implement a GA and demonstrate its functionality in solving the chosen problem.
4. Apply modifications/variations for overcoming the challenges in the implemented solution.
5. Implement a GA and demonstrate its functionality in solving the chosen problem.
6. Modify GA parameters (e.g., population size, mutation rate) and operators (e.g., different crossover techniques) to observe their impact on performance.
7. Optimize the GA performance through parameter tuning and by experimenting with different variations.

EXPLANATION / SOLUTIONS (DESIGN):

Code:

```
# Genetic Algorithm for Job Scheduling on Machines


import random

import numpy as np

import matplotlib.pyplot as plt
```

```
# Define the problem: list of jobs with processing times
jobs = [10, 20, 30, 25, 15]  # Job processing times
num_machines = 3  # Number of machines

# Fitness function: calculates the makespan (maximum time any machine takes to finish)
def fitness(chromosome):
    machine_times = [0] * num_machines
    for job, machine in enumerate(chromosome):
        machine_times[machine] += jobs[job]
    return -max(machine_times)  # Minimize the maximum time, hence negative for GA

# Generate a random initial population of chromosomes
def generate_population(population_size, chromosome_length):
    return [np.random.randint(num_machines, size=chromosome_length).tolist() for _ in
range(population_size)]

# Selection process using tournament selection
def selection(population, fitness_scores):
    tournament_size = 3
    selected = random.sample(list(zip(population, fitness_scores)), tournament_size)
    selected = max(selected, key=lambda x: x[1])  # Choose the individual with the highest
fitness
    return selected[0]

# Crossover (one-point crossover)
def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    return parent1[:point] + parent2[point:], parent2[:point] + parent1[point:]
```

```
# Mutation (randomly assigns a job to a different machine)
def mutate(chromosome, mutation_rate=0.1):
    for i in range(len(chromosome)):
        if random.random() < mutation_rate:
            chromosome[i] = random.randint(0, num_machines - 1)  # Reassign job to a
random machine
    return chromosome


# Genetic Algorithm
def genetic_algorithm(population_size, chromosome_length, generations, mutation_rate):
    population = generate_population(population_size, chromosome_length)
    best_fitness = []


    for generation in range(generations):
        fitness_scores = [fitness(chromosome) for chromosome in population]


        # Track the best solution in each generation
        best_fitness.append(-max(fitness_scores))  # Convert back to positive makespan for
tracking


        new_population = []
        for _ in range(population_size // 2):
            parent1 = selection(population, fitness_scores)
            parent2 = selection(population, fitness_scores)


            # Crossover
            offspring1, offspring2 = crossover(parent1, parent2)


            # Mutation
            offspring1 = mutate(offspring1, mutation_rate)
            offspring2 = mutate(offspring2, mutation_rate)

            new_population.extend([offspring1, offspring2])

        population = new_population

    # Find the best chromosome in the final population
    final_fitness_scores = [fitness(chromosome) for chromosome in population]
    best_chromosome = population[final_fitness_scores.index(max(final_fitness_scores))]

    return best_chromosome, best_fitness

# Run the genetic algorithm
population_size = 10
chromosome_length = len(jobs)
generations = 50
mutation_rate = 0.1

best_solution, fitness_history = genetic_algorithm(population_size, chromosome_length,
generations, mutation_rate)

# Print the best solution and its fitness value (makespan)
print("Best solution (job to machine assignment):", best_solution)
makespan = -fitness(best_solution)  # Best makespan (minimized)
print("Best makespan:", makespan)

# Plot the fitness (makespan) over generations
plt.plot(fitness_history)
```

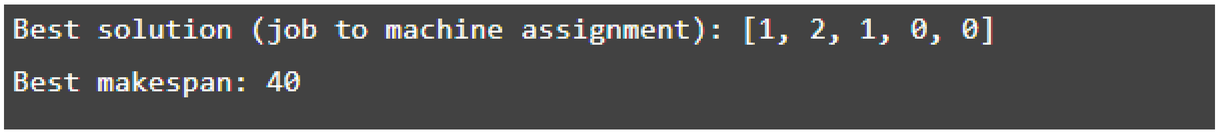
plt.title("Makespan over Generations")

plt.xlabel("Generation")

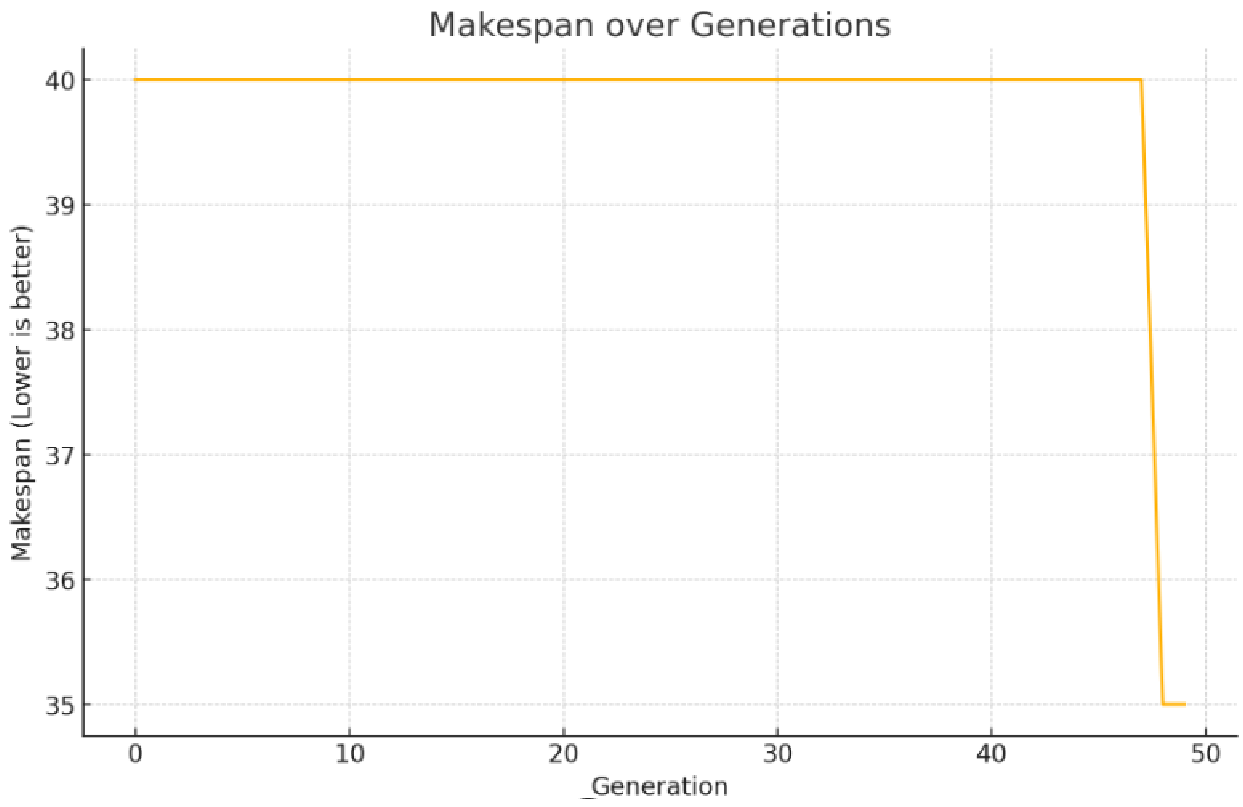
plt.ylabel("Makespan (Lower is better)")

plt.show()

Output:



```
Best solution (job to machine assignment): [1, 2, 1, 0, 0]
Best makespan: 40
```


Makespan over Generations
Makespan (Lower is better)
_Generation