DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING

Course Code: CS-323

Course Title: ***ARTIFICIAL INTELLIGENCE***

Open Ended Lab

TE Batch 2022, Fall Semester 2024

GRADING RUBRIC

GROUP MEMBERS:

Student No.	Name	Roll No.
S1	HEMAT	CS-22115
S2	HADI RAZA	CS-22133
S3	ROSHIK ADIL	CS-22143

<table>
<tr><th colspan="4" rowspan="2">CRITERIA AND SCALES</th><th colspan="3">Marks Obtained</th></tr>
<tr><th>S1</th><th>S2</th><th>S3</th></tr>
<tr><td colspan="4">Criterion 1: Has the student appropriately simulated the working of the genetic algorithm?</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>-</td></tr>
<tr><td>The explanation is too basic.</td><td>The algorithm is explained well with an example.</td><td>The explanation is much more comprehensive.</td><td></td></tr>
<tr><td colspan="4">Criterion 2: How well is the student's understanding of the genetic algorithm?</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>The student has no understanding.</td><td>The student has a basic understanding.</td><td>The student has a good understanding.</td><td>The student has an excellent understanding.</td></tr>
<tr><td colspan="4">Criterion 3: How good is the programming implementation?</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>The project could not be implemented.</td><td>The project has been implemented partially.</td><td>The project has been implemented completely but can be improved.</td><td>The project has been implemented completely and impressively.</td></tr>
<tr><td colspan="4">Criterion 4: How good is the selected application?</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>-</td></tr>
<tr><td>The chosen application is too simple.</td><td>The application is fit to be chosen.</td><td>The application is different and impressive.</td><td></td></tr>
<tr><td colspan="4">Criterion 5: How well-written is the report?</td><td rowspan="3"></td><td rowspan="3"></td><td rowspan="3"></td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>-</td></tr>
<tr><td>The submitted report is unfit to be graded.</td><td>The report is partially acceptable.</td><td>The report is complete and concise.</td><td></td></tr>
<tr><td colspan="4" align="right">Total Marks:</td><td></td><td></td><td></td></tr>
</table>

GENETIC ALGORITHM FOR TASK SCHEDULING

INTRODUCTION:

In this report, we present the implementation of a **Genetic Algorithm (GA)** to solve a **Task Scheduling** problem, which aims to distribute tasks across multiple processors in a way that minimizes the maximum load on any processor. The GA mimics the process of natural selection, applying genetic operators like **selection**, **crossover**, and **mutation** to evolve a population of solutions toward an optimal or near-optimal solution.

PROBLEM DESCRIPTION:

The problem is to schedule a set of tasks, each with a specified duration, across a given number of processors. The goal is to minimize the **maximum load** on any processor, ensuring that the most loaded processor has the smallest possible load.

GENETIC ALGORITHM OVERVIEW:

Genetic Algorithms are inspired by the process of natural selection in biology. The basic steps in a GA are:

- **Initialization:** A population of random solutions is generated.
- **Selection:** Parents are selected based on their fitness.
- **Crossover (Recombination):** Parents combine their genetic material to produce offspring.
- **Mutation:** Random changes are applied to offspring to maintain diversity.
- **Replacement:** The next generation is formed, and the process repeats for a set number of generations or until convergence is reached.

The Genetic Algorithm for the task scheduling problem works as follows:

- **Chromosome Representation:** Each individual (chromosome) in the population represents a task assignment, where each gene (value) indicates which processor aspecific task is assigned to.
- **Fitness Function:** The fitness of each solution is evaluated based on how evenly tasks are distributed across processors. The goal is to minimize the maximum load on any processor.
- **Selection:** Tournament selection is used to choose parents based on their fitness.
- **Crossover:** A single-point crossover is applied to combine the genetic material of two parents.
- **Mutation:** A mutation operator randomly changes a task's assignment to a different processor with a given probability.

CODE IMPLEMENTATION:

The following Python code implements the Genetic Algorithm for the Task Scheduling problem:

- **FITNESS FUNCTION:**

```
def fitness_function(chromosome, tasks, num_processors):
    processor_loads = [0] * num_processors
    for i, task in enumerate(tasks):
        processor_loads[chromosome[i]] += task
    max_load = max(processor_loads)
    baseline = sum(tasks) # Worst-case load if all tasks are assigned to one processor
    return baseline - max_load  # Shift fitness to ensure non-negative values
```

❖ **EXPLANATION:**

  - The **fitness function** calculates how well a chromosome distributes tasks across processors. It assigns each task to a processor and calculates the load for each processor.
  - The **fitness score** is defined as the difference between the baseline (worst-case load) and the maximum load across processors. The higher the fitness, the better the solution, as it represents a more balanced task distribution.

- **GENERATE INITIAL POPULATION:**

```
def generate_population(size, num_tasks, num_processors):
    return [[random.randint(0, num_processors - 1) for _ in range(num_tasks)] for _ in
range(size)]
```

❖ **EXPLANATION:**

  - The function generates an initial **population** of random chromosomes, where each chromosome is a list of task assignments. Each task is assigned randomly to one of the processors.

- **SELECTION (TOURNAMENT):**

```
def select_parents(population, fitness):
    selected = random.choices(population, weights=[f + 1 for f in fitness], k=2)
    return selected
```

❖ **EXPLANATION:**

  - **Tournament selection** is used to select two parents. The probability of an individual being selected is based on its fitness, with better individuals having a higher chance of being selected.

- **CROSSOVER (SINGLE POINT):**

```
def crossover(parent1, parent2):
```

```
    point = random.randint(1, len(parent1) - 1)
    offspring1 = parent1[:point] + parent2[point:]
    offspring2 = parent2[:point] + parent1[point:]
    return offspring1, offspring2
```

- ❖ **EXPLANATION:**
  - **Single-point crossover** combines two parent chromosomes at a randomly chosen point. It produces two offspring by exchanging the genetic material from bothparents.

- **MUTATION:**

```
def mutate(chromosome, mutation_rate, num_processors):
    if random.random() < mutation_rate:
        idx = random.randint(0, len(chromosome) - 1)
        chromosome[idx] = random.randint(0, num_processors - 1)
    return chromosome
```

- ❖ **EXPLANATION:**
  - **Mutation** introduces randomness into the algorithm by randomly changing the processor assignment of a task. The mutation rate controls how often this happens.

- **MAIN GENETIC ALGORITHM EXECUTION:**

```
def genetic_algorithm(tasks, num_processors, pop_size, generations, mutation_rate):
    num_tasks = len(tasks)
    population = generate_population(pop_size, num_tasks, num_processors)
    print("\nInitial Population:")
    for i, p in enumerate(population):
        print(f"Chromosome {i + 1}: {p}")

    for generation in range(generations):
        fitness = [fitness_function(chromosome, tasks, num_processors) for chromosome in
population]
        print(f"\nGeneration {generation + 1}: Best Fitness = {max(fitness)}")

        if max(fitness) <= 0:
            raise ValueError("Fitness values are invalid. Check input tasks and processor
count.")

        new_population = []
        for _ in range(pop_size // 2):
            parent1, parent2 = select_parents(population, fitness)
            offspring1, offspring2 = crossover(parent1, parent2)
            offspring1 = mutate(offspring1, mutation_rate, num_processors)
            offspring2 = mutate(offspring2, mutation_rate, num_processors)
```

```
            new_population.extend([offspring1, offspring2])
        population = new_population

    best_solution = max(population, key=lambda c: fitness_function(c, tasks,
num_processors))
    return best_solution, sum(tasks) - fitness_function(best_solution, tasks,
num_processors)
```

❖ **EXPLANATION:**

- This function runs the Genetic Algorithm for a specified number of generations.
- In each generation, the fitness of the current population is evaluated, and the best solutions are selected and evolved using crossover and mutation.
- The best solution after all generations is returned, along with the minimum maximum **load**.

- **MAIN EXECUTION:**

```
if__name__== "__main__":
    print("=== Genetic Algorithm: Task Scheduling ===")
    tasks = list(map(int, input("Enter task durations separated by spaces: ").split()))
    num_processors = int(input("Enter the number of processors: "))
    pop_size = int(input("Enter population size: "))
    generations = int(input("Enter number of generations: "))
    mutation_rate = float(input("Enter mutation rate (0 to 1): "))

    try:
        best_solution, best_fitness = genetic_algorithm(
            tasks=tasks,
            num_processors=num_processors,
            pop_size=pop_size,
            generations=generations,
            mutation_rate=mutation_rate
        )
        print("\n=== Final Results ===")
        print(f"Best Task Assignment: {best_solution}")
        print(f"Minimum Maximum Load: {best_fitness}")
    except ValueError as e:
        print(f"Error: {e}")
```

❖ **EXPLANATION:**

- The program prompts the user to input the task durations, number of processors, population size, number of generations, and mutation rate.
- The **best solution** and **minimum maximum load** are then printed after running the Genetic Algorithm.

EXAMPLE:

For example, consider the following inputs:

- ✓ Tasks: [8, 4, 3, 9, 10]
- ✓ Number of processors: 4
- ✓ Population size: 10
- ✓ Generations: 20
- ✓ Mutation rate: 0.1

The **Output** would be something like:

Best Task Assignment: [3, 3, 0, 1, 0]Minimum

Maximum Load: 13

```
Generation 16: Best Fitness = 21

Generation 17: Best Fitness = 21

Generation 18: Best Fitness = 21

Generation 19: Best Fitness = 21

Generation 20: Best Fitness = 21

=== Final Results ===
Best Task Assignment: [3, 3, 0, 1, 0]
Minimum Maximum Load: 13
```

This means that after running the algorithm, the tasks have been assigned to processors in away that minimizes the maximum load to **13**.

CONCLUSION:

The Genetic Algorithm successfully solves the task scheduling problem by assigning tasks to processors in a way that minimizes the maximum load. Through the use of selection, crossover, and mutation, the GA is able to find near-optimal solutions. However, as with most evolutionary algorithms, the quality of the solution depends on factors such as population size, number of generations, and mutation rate.

FUTURE WORK:

Possible improvements could include:

- ✓ Experimenting with different mutation and crossover strategies.
- ✓ Adding elitism to retain the best solutions across generations.
- ✓ Parallelizing the algorithm for faster computation with larger task sets