EAST WEST UNIVERSITY

"Assignment-2: Robot Task Optimization Using Genetic Algorithm"

Course Code : CSE366
Course Title : Artificial Intelligence
Section : 03

Submitted To:

Instructor:

Dr. Mohammad Rifat Ahmmad Rashid
Assistant Professor,
Department of Computer Science & Engineering

Submitted By:

NAME: A.B.M. Ilman Farabi
ID : 2021-3-60-111

Assignment -2 Title: Robot Task Optimization Using Genetic Algorithm

Objective: The goal of this assignment is to develop and implement a Genetic Algorithm (GA) to optimize the assignment of multiple robots to a set of tasks in a dynamic production environment. Your primary objectives are to minimize the total production time, ensure a balanced workload across robots, and prioritize critical tasks effectively. Additionally, you will create a detailed visualization to illustrate the final task assignments, robot efficiencies, and task priorities.

Overview:

Data Preparation: Generate mock data for tasks (including durations and priorities) and robots (including efficiency factors).

```
# Function to generate mock data for tasks and robots
def generate_mock_data(num_tasks=10, num_robots=5):
    task_durations = [x for x in range(11)]  # Task durations
    task_priorities = [x for x in range(11)]  # Task priorities
    robot_efficiencies = [0.1, 0.01, 0.2, 0.3, 0.4]  # Robot
efficiencies
    return task_durations, task_priorities, robot_efficiencies
```

GA Implementation: Implement a Genetic Algorithm to optimize task assignments considering task duration, robot efficiency, and task priority.

```
# GA algorithm implementation
def run_genetic_algorithm(task_durations, task_priorities,
robot_efficiencies):
    population_size = 50
    n_generations = 100

    population = [np.random.randint(0, len(robot_efficiencies),
size=len(task_durations)) for _ in range(population_size)]
    best_solution = None
    best_fitness = float('-inf')

    for _ in range(n_generations):
        # Evaluate fitness for each individual in the population
        fitness_values = [calculate_fitness(sol, task_durations,
task_priorities, robot_efficiencies) for sol in population]

        # Select parents for crossover using tournament selection
        selected_parents = [tournament_selection(population,
fitness_values, tournament_size=5) for _ in range(population_size //
2)]

        # Perform crossover to generate offspring
        offspring_population = [single_point_crossover(parents) for
parents in selected_parents]
```

```
        offspring_population = [child for pair in offspring_population
for child in pair]  # Flatten list of offspring

        # Apply mutation to the offspring
        offspring_population = [mutation(child, mutation_rate=0.1) for
child in offspring_population]

        # Combine parents and offspring to form the next generation
        population = offspring_population

        # Find the best solution in the current generation
        for sol, fitness in zip(population, fitness_values):
            if fitness > best_fitness:
                best_solution = sol
                best_fitness = fitness

    return best_solution
```

Visualization: Create a grid visualization of the task assignments highlighting key information.

```
# Improved visualization function
def visualize_assignments_improved(solution, task_durations,
task_priorities, robot_efficiencies):
    # Create a grid for visualization based on the solution provided
    grid = np.zeros((len(robot_efficiencies), len(task_durations)))
    for task_idx, robot_idx in enumerate(solution):
        grid[robot_idx, task_idx] = task_durations[task_idx]

    fig, ax = plt.subplots(figsize=(12, 6))
    cmap = mcolors.LinearSegmentedColormap.from_list("", ["white",
"blue"])  # Changed color intensity to blue

    # Display the grid with task durations
    cax = ax.matshow(grid, cmap=cmap)
    fig.colorbar(cax, label='Task Duration (hours)')

    # Annotate each cell with task priority and duration
    for i in range(len(robot_efficiencies)):
        for j in range(len(task_durations)):
            ax.text(j, i, f'{task_durations[j]} hr\n(Prio
{task_priorities[j]})',  # Changed display format
                    ha='center', va='center', color='black')

    # Set the ticks and labels for tasks and robots
    ax.set_xticks(np.arange(len(task_durations)))
    ax.set_yticks(np.arange(len(robot_efficiencies)))
    ax.set_xticklabels([f'Task {i+1}' for i in
range(len(task_durations))], rotation=45, ha="left")
```

```
    ax.set_yticklabels([f'Robot {i+1} (Efficiency: {eff:.2f})' for i,
eff in enumerate(robot_efficiencies)])

    plt.xlabel('Tasks')
    plt.ylabel('Robots')
    plt.title('Task Assignments with Task Duration and Priority')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    num_tasks = 10
    num_robots = 3
    task_durations, task_priorities, robot_efficiencies =
generate_mock_data(num_tasks, num_robots)

    # Run GA to find the best solution
    best_solution = run_genetic_algorithm(task_durations,
task_priorities, robot_efficiencies)

    # Visualize the best solution
    visualize_assignments_improved(best_solution, task_durations,
task_priorities, robot_efficiencies)
```

Explanation-

1. Fitness Function

The fitness function assesses the quality of each potential solution (task assignment) based on two main criteria: minimizing total production time and workload balance across robots. Here's how it's implemented:

- **Total Production Time (Total):** The total production time is calculated by determining the time each robot spends on its assigned tasks and then selecting the maximum time among all robots. This is achieved by iterating through each task and summing the durations considering the robot's efficiency.
- **Workload Balance (B):** Workload balance measures the variation in production time among different robots. It is computed as the standard deviation of the total times across all robots.
- **Fitness Function (F):** The fitness function combines the total production time and workload balance to evaluate the quality of a solution. It aims to minimize both metrics, and it's defined as the inverse of the sum of Ttotal and B.

2. Selection

In the genetic algorithm, the selection process determines which individuals (task assignments) from the current population will be chosen as parents for producing the next generation. The tournament selection method is used here:

- **Tournament Selection:** For each pair of parents to be selected, a small subset of individuals (tournament) is randomly chosen from the population. The fitness of each individual in the tournament is evaluated, and the one with the highest fitness is chosen as a parent for crossover.

3. Crossover

Crossover is a genetic operation that combines genetic information from two parents to create offspring with potentially better characteristics. Here, a single-point crossover method is employed:

- **Single-Point Crossover:** A random crossover point is selected along the chromosome (task assignment vector). Offspring are created by swapping the genetic information beyond this crossover point between the two parents.

4. Mutation

Mutation introduces random variations into the population, ensuring genetic diversity and preventing premature convergence to suboptimal solutions:

- **Task Swapping Mutation:** With a certain probability (mutation rate), two tasks within an individual's assignment list are randomly selected, and their assigned robots are swapped.

5. Visualization

The visualization function creates a grid representation of the task assignments, task durations, and task priorities. It provides a visual understanding of how tasks are distributed among robots and their corresponding durations and priorities:

- **Grid Visualization:** Each row represents a robot, and each column represents a task. The intensity of color in each cell reflects the duration of the corresponding task, with annotations indicating the task's priority.
- **Robot Efficiency and Task Priority Annotation:** The row labels display the efficiency of each robot, while the column labels indicate the priority of each task.

Source code:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import matplotlib.patches as mpatches

# Function to generate mock data for tasks and robots
def generate_mock_data(num_tasks=10, num_robots=5):
    task_durations = [x for x in range(11)]  # Task durations
    task_priorities = [x for x in range(11)]  # Task priorities
    robot_efficiencies = [0.1, 0.01, 0.2, 0.3, 0.4]  # Robot 
efficiencies
    return task_durations, task_priorities, robot_efficiencies
```

```
# Placeholder for the fitness function calculation
def calculate_fitness(solution, task_durations, task_priorities,
robot_efficiencies):
    # Calculate total production time for each robot
    robot_times = np.zeros(len(robot_efficiencies))
    for task_idx, robot_idx in enumerate(solution):
        robot_times[robot_idx] += task_durations[task_idx] /
robot_efficiencies[robot_idx]

    # Total production time is the maximum time any robot takes to
complete its tasks
    total_production_time = np.max(robot_times)

    # Workload balance
    workload_balance = np.std(robot_times)

    # Fitness function: minimize total production time and workload
balance
    fitness = 1 / (total_production_time + workload_balance)

    return fitness

# Placeholder for the selection process
def tournament_selection(population, fitness_values, tournament_size):
    selected_parents = []
    for _ in range(2):  # Select 2 parents
        tournament_indices = np.random.choice(len(population),
size=tournament_size, replace=False)
        tournament_fitness = [fitness_values[i] for i in
tournament_indices]
        winner_index =
tournament_indices[np.argmax(tournament_fitness)]
        selected_parents.append(population[winner_index])
    return selected_parents

# Placeholder for the crossover operation
def single_point_crossover(parents):
    crossover_point = np.random.randint(1, len(parents[0]))  # Choose a
random crossover point
    child1 = np.concatenate((parents[0][:crossover_point],
parents[1][crossover_point:]))
    child2 = np.concatenate((parents[1][:crossover_point],
parents[0][crossover_point:]))
    return child1, child2

# Placeholder for the mutation operation
def mutation(solution, mutation_rate):
```

```
    if np.random.rand() < mutation_rate:
        idx1, idx2 = np.random.choice(len(solution), size=2,
replace=False)
        solution[idx1], solution[idx2] = solution[idx2], solution[idx1]
    return solution

# GA algorithm implementation
def run_genetic_algorithm(task_durations, task_priorities,
robot_efficiencies):
    population_size = 50
    n_generations = 100

    population = [np.random.randint(0, len(robot_efficiencies),
size=len(task_durations)) for _ in range(population_size)]
    best_solution = None
    best_fitness = float('-inf')

    for _ in range(n_generations):
        # Evaluate fitness for each individual in the population
        fitness_values = [calculate_fitness(sol, task_durations,
task_priorities, robot_efficiencies) for sol in population]

        # Select parents for crossover using tournament selection
        selected_parents = [tournament_selection(population,
fitness_values, tournament_size=5) for _ in range(population_size //
2)]

        # Perform crossover to generate offspring
        offspring_population = [single_point_crossover(parents) for
parents in selected_parents]
        offspring_population = [child for pair in offspring_population
for child in pair]  # Flatten list of offspring

        # Apply mutation to the offspring
        offspring_population = [mutation(child, mutation_rate=0.1) for
child in offspring_population]

        # Combine parents and offspring to form the next generation
        population = offspring_population

        # Find the best solution in the current generation
        for sol, fitness in zip(population, fitness_values):
            if fitness > best_fitness:
                best_solution = sol
                best_fitness = fitness

    return best_solution
```

```python
# Improved visualization function
def visualize_assignments_improved(solution, task_durations,
task_priorities, robot_efficiencies):
    # Create a grid for visualization based on the solution provided
    grid = np.zeros((len(robot_efficiencies), len(task_durations)))
    for task_idx, robot_idx in enumerate(solution):
        grid[robot_idx, task_idx] = task_durations[task_idx]

    fig, ax = plt.subplots(figsize=(12, 6))
    cmap = mcolors.LinearSegmentedColormap.from_list("", ["white",
"blue"])  # Changed color intensity to blue

    # Display the grid with task durations
    cax = ax.matshow(grid, cmap=cmap)
    fig.colorbar(cax, label='Task Duration (hours)')

    # Annotate each cell with task priority and duration
    for i in range(len(robot_efficiencies)):
        for j in range(len(task_durations)):
            ax.text(j, i, f'{task_durations[j]} hr\n(Prio
{task_priorities[j]})',  # Changed display format
                    ha='center', va='center', color='black')

    # Set the ticks and labels for tasks and robots
    ax.set_xticks(np.arange(len(task_durations)))
    ax.set_yticks(np.arange(len(robot_efficiencies)))
    ax.set_xticklabels([f'Task {i+1}' for i in
range(len(task_durations))], rotation=45, ha="left")
    ax.set_yticklabels([f'Robot {i+1} (Efficiency: {eff:.2f})' for i,
eff in enumerate(robot_efficiencies)])

    plt.xlabel('Tasks')
    plt.ylabel('Robots')
    plt.title('Task Assignments with Task Duration and Priority')

    plt.tight_layout()
    plt.show()

if __name__ == "__main__":
    num_tasks = 10
    num_robots = 3
    task_durations, task_priorities, robot_efficiencies =
generate_mock_data(num_tasks, num_robots)

    # Run GA to find the best solution
    best_solution = run_genetic_algorithm(task_durations,
task_priorities, robot_efficiencies)
```

```
    # Visualize the best solution
    visualize_assignments_improved(best_solution, task_durations,
task_priorities, robot_efficiencies)
```

Output

```
Task Priorities:
----------------
Task 1: Priority 0
Task 2: Priority 1
Task 3: Priority 2
Task 4: Priority 3
Task 5: Priority 4
Task 6: Priority 5
Task 7: Priority 6
Task 8: Priority 7
Task 9: Priority 8
Task 10: Priority 9
```