EAST WEST UNIVERSITY

Department of Computer Science and Engineering

Course Code: CSE366

Course Title: Artificial Intelligence

Section: 04

Semester: Spring 24

Assignment- 02

Submitted to:

Dr. Mohammad Rifat Ahmmad Rashid

Assistant Professor

Department of Computer Science and Engineering

East West University

Submitted by:

Name: B M Shahria Alam

ID: 2021-3-60-016

Date of submission: 4th April, 2024.

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.

Detailed Requirements

1. Background:

- You have a set of tasks, each with a specified duration and priority.
- A pool of robots is available, each with a unique efficiency factor.
- The production environment is dynamic, with tasks and priorities potentially changing over time

2. Tasks:

- **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 = np.random.randint(1, 11, size=num_tasks)  # Random task durations between 1 and 10 hours
    task_priorities = np.random.randint(1, 6, size=num_tasks)  # Random task priorities between 1 and 5
    robot_efficiencies = np.random.uniform(0.5, 1.5, size=num_robots)  # Random robot efficiencies between 0.5 and 1.5
    return task_durations, task_priorities, robot_efficiencies
```

Here we create random duration, priorities of num_task where num_tasks=10, it can be any value which need to be solved by the robots. we also define each robot efficiencies of num_robots which is currently defined 5 but can be pass anything through the parameter.

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

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

3. Genetic Algorithm Components:

• Individual Representation: Represent each potential solution as a vector where each element indicates the robot assigned to each task.

Individual I is represented as a vector of N integers, where N is the number of tasks, and each integer In (where 1≤n≤N) corresponds to the ID of the robot assigned to task n. I=[r1,r2,...,rN]

First we defined initial population randomly by,

```python
# GA algorithm placeholder
def run_genetic_algorithm(task_durations, task_priorities, robot_efficiencies):
    population_size = 50
    generation = 100
    mutation_rate = 0.1

    # Placeholder for the initial population generation
    population = [np.random.randint(0, len(robot_efficiencies), size=len(task_durations)) for _ in range(population_size)]
    for _ in range(generation):
        fitness = fitness_function(population, task_durations, task_priorities, robot_efficiencies) # Placeholder for the f
        parents = select_parents(population, fitness) # Placeholder for the selection process
        offspring = crossover(parents, int(population_size / 2 )) # Placeholder for the crossover
        population = mutation(offspring, mutation_rate, robot_efficiencies) # Placeholder for the mutation operation

    best_solution = population[np.argmax(fitness)] #print(fitness[np.argmax(fitness)])

    return best_solution
```

• **Fitness Function:** The fitness function aims to minimize the total production time while ensuring a balanced workload across robots and prioritizing critical tasks. It can be decomposed into several components: Total Time, Workload balance;

- • Calculate the total production time, Ttotal, as the maximum time taken by any robot based on its assigned tasks and efficiency.
- • Compute workload balance, B, as the standard deviation of the total times across all robots.
- • Define the fitness function, F, to minimize both Ttotal and B, incorporating task priorities.

```python
def fitness_function(population, task_durations, task_priorities, robot_efficiencies):
    fitness =[] #initialize
    total_robot = len(robot_efficiencies) #Calculates the total number of robots
    for i in range(len(population)):
        present_population = population[i]
        Tr = np.zeros(total_robot, dtype=int) #Initializes a NumPy array
        for j in range(len(present_population)):
            task = j
            robot = present_population[task]
            Task = task_durations[task]       #Retrieves the duration of the current task.
            priority = task_priorities[task]  #Retrieves the priority of the current task.
            effen = robot_efficiencies[robot] #Retrieves the efficiency of the robot
            Tr[robot] = Tr[robot] + ((Task * priority) / effen) #Fitness function formula
        Ttotal = np.max(Tr) #Calculates the maximum time taken by any robot to complete tasks.
        B = np.std(Tr)      #Standard deviation
        fitness.append(1/(Ttotal + B))
    return fitness
```

this is our fitness function, where we implemented

Tr = ∑ n∈tasks(r) Dn*Pn / Er

Ttotal = max (T1, T2, ... , TR)

where:

• tasks(r) is the set of tasks assigned to robot r,

• Dn is the duration of task n,

• Pn is the priority weight of task n,

• Er is the efficiency of robot R

• R is the total number of robots.

B=σ(T1,T2,...,TR) here this is the standard davition for balacing the workload,

then F(I) = Ttota + B

4. • Selection, Crossover, and Mutation:

Implemented Uniform Cost Search (UCS) and A* (A Star) pathfinding algorithms. The selection process is crucial for guiding the GA towards optimal solutions by choosing individuals from the current population to breed the next generation. We have used Tournament Selection. Where we have choose half of the population to be in mating pool by their fitness value, we choose the values whom's fitness value Is the lowest because we are trying to minimize the cost and minimize the standard deviation of workload

```
def crossover(parents, num_offspring):
    # Single point crossover
    offspring = []
    for _ in range(num_offspring):
        crossover_point = np.random.randint(1, len(parents[0]))   #Generates a r
        parent1_idx = np.random.randint(0, len(parents))
        parent2_idx = np.random.randint(0, len(parents))          #Randomly sele
        offspring_part1 = parents[parent1_idx][0:crossover_point] #Splits the va
        offspring_part2 = parents[parent2_idx][crossover_point:]
        offspring2_part1 = parents[parent2_idx][0:crossover_point]#Combines the
        offspring2_part2 = parents[parent1_idx][crossover_point:]
        offspring.append(np.concatenate((offspring_part1, offspring_part2)))
        offspring.append(np.concatenate((offspring2_part1, offspring2_part2)))

    return offspring
```

```
def mutation(offspring, mutation_rate, robot_efficiencies):
    # Mutation changes a single value in each offspring randomly.
    for idx in range(len(offspring)):
        for _ in range(int(len(offspring[idx])*mutation_rate)):
            First_num = np.random.randint(0, len(offspring[idx]))  #Generate two random variable
            Second_num = np.random.randint(0, len(offspring[idx]))
            offspring[idx][First_num], offspring[idx][Second_num] = offspring[idx][Second_num] , offspring[idx][First_num]
    return offspring
```

```
def select_parents(population, fitness):
    num_parents = int(len(population)/2) #Calculates the number of parents
    parents = []

    for _ in range(num_parents):
        max_fitness_idx = np.argmax(fitness) #Finds the highest fitness value in the fitness list (individual).
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -np.inf  # so this individual is not selected again. Set the value ti - infinity

    return parents
```

5. Visualization

```
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"])  # Custom colormap

    # 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(task_durations)):
        for j in range(len(robot_efficiencies)):
            text_color = 'white' if grid[j, i] > 0 and task_durations[i] >= 5 else 'black'
            ax.text(i, j, f'P{task_priorities[i]}\n{task_durations[i]}H', va='center', ha='center', color=text_color)


    # 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')

    # Create a legend for task priorities
    priority_patches = [mpatches.Patch(color='white', label=f'Priority {i}') for i in range(1, 6)]
    plt.legend(handles=priority_patches, bbox_to_anchor=(1.20, 1), loc='upper left', title="Task Priorities")

    plt.tight_layout()
    plt.show()
```