

**“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

# 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**

A screenshot of a computer program

Description automatically generated

A graph with blue squares

Description automatically generated