Name: S.M.NAZMUL HASAN
ID: 2021-2-60-040
Assignment: 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.

Assignment Details

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)
    task_priorities = np.random.randint(1, 6, size=num_tasks)
    robot_efficiencies = [0.1,0.0002,0.3,0.4,0.5 ]
    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.

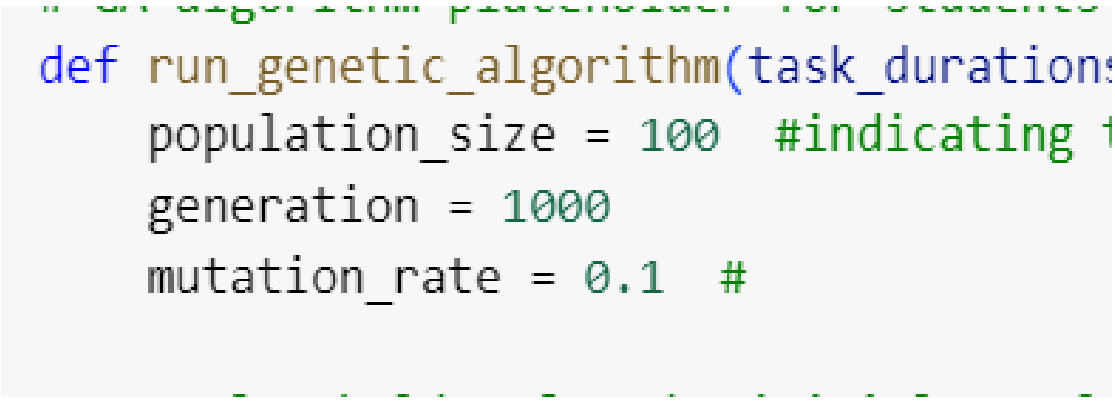
- **GA Implementation**: Implement a Genetic Algorithm to optimize task assignments considering task duration, robot efficiency, and task priority.
- Visualization: Create a grid visualization of the task assignments highlighting key information.

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,
Here we define population size as 100, and generation 1000, we have tried with different values but 1000 seems fine.

Here population array will have 100 np array inside it, where each array will have values ranging from 0 to how many robot we have in this case 5, sinch the counting start from 0, it will end at 4 and each inside array will have 10 values**. [0,4,2,2,4,3,0,3,3,2]**

- **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.
- Check the part fitness function calculations.

```python
def fitness_function(population, task_durations, task_priorities, robot_efficiencies):
    fitness =[]
    total_robot = len(robot_efficiencies)

    for i in range(len(population)):
        present_population = population[i] # taking each Individual
        Tr = np.zeros(total_robot, dtype=int) # array filled with zeros for: total time tak
        #here, Tr= total production time


        for j in range(len(present_population)):
            task = j  # taking each number of each Individual
            robot = present_population[task]
            td = task_durations[task]
            tp = task_priorities[task]
            re = robot_efficiencies[robot]
            Tr[robot] = Tr[robot] + ((td * tp) / re)
        Ttotal = np.max(Tr)


        # Workload Balance (B)
        B = np.std(Tr)

        # fitness.append((Ttotal + B))
        fitness.append(1/(Ttotal + B))  #inverse


    return fitness
```

- this is our fitness function, where we implemented

$Tr = \sum_{n \in tasks(r)} Dn*Pn / Er$

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

Here we have experimented with 2 things, if we only return Ttotal + B as fitness function, we need to select according to minimum fitness value. but if we take inverse of (Ttotal + B) = 1/ (Ttotal + B) then we can take the maximum as

- **Selection, Crossover, and Mutation:**

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 choosed half of the population to be in mating pool by their fitness value, we choosed the values whom's fitness value Is the lowest. because we are trying to minimize the cost and minimize the standard deviation of workload.

so again if we inverse the fitness value, we can take the maximum fintess function to select parents.

```
def select_parents(population, fitness):

    select = np.random.randint(1, int(len(population)/2))  # 1 to t

    parents = []

    for _ in range(select):
        max_fitness_idx = np.argmax(fitness)  # index of the maximu
        parents.append(population[max_fitness_idx])
        fitness[max_fitness_idx] = -np.inf  #  ensures that the sel

    return parents
```

after selecing each parent we have made the fitness value of them infinite so that they would never piked up again.

Crossover:

Crossover is a genetic operation used to combine the genetic information of two parents to generate new offspring. It is pivotal for introducing new genetic combinations into the population.

we have used single point crossover.

Single-Point Crossover:

• Randomly choose a crossover point on the parent individuals' genomes.

• Create offspring by swapping all genes (task assignments in our context)

after this point between the two parents.

• Example: If our crossover point is 3 and we have two parents

[A, B, C, D, E] and [V, W, X, Y, Z], the offspring would be
[A, B, C, Y, Z] and [V, W, X, D, E]. we have tried another
implementation of this,

```
def crossover(p, num_chield):    # num_chield = population_size / 2


    chield = []
    for _ in range(num_chield):

        crossover_point = np.random.randint(1, len(p[0]))

        p1 = np.random.randint(0, len(p))
        p2 = np.random.randint(0, len(p))

        c11 = p[p1][0:crossover_point]  # 0 theke crossover_point
        c12 = p[p2][crossover_point:]   # crossover_point theke last

        c21 = p[p2][0:crossover_point]
        c22 = p[p1][crossover_point:]

        chield.append(np.concatenate((c11, c12)))
        chield.append(np.concatenate((c21, c22)))



    return chield
```

we have tried to use 50% parent and only make offspring of 50% then make them new generation. so in our first implementation we choosed 2 parent randomly each time and make 1 offspring each time.

and in our 2nd implementation we made 2 offspring from 2 parent by swaping their component. on each iterations, we have choosed the random crossover_point.

Mutation:

Mutation introduces random genetic variations, providing new genetic structures for exploration and helping the algorithm escape local optima.

1. Task Swapping: Randomly select two tasks within an individual's assignment list and swap their assigned robots.

another implementation is, randomly choose a point and put random value between 0 to len of robots. each gives the different output at the end. we finalized the task swapping.

```
def mutation(chield, mutation_rate):

    for idx in range(len(chield)):
        for _ in range(int(len(chield[idx])*mutation_rate)):
            T1 = np.random.randint(0, len(chield[idx]))
            T2 = np.random.randint(0, len(chield[idx]))
            chield[idx][T1]= chield[idx][T2]
            chield[idx][T2]= chield[idx][T1]
            # chield[idx][T1], chield[idx][T2] = chield[idx][T2] , chield[idx][T1]
    return chield
```

Choosing Best solution:

```
fitness = fitness_function(population, task_durations, task_priorities, robot_efficiencies)


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

print("Fitness: ",fitness[np.argmax(fitness)])
best_solution = population[np.argmax(fitness)]

return best_solution
```

after the last mutation we have calculated fitness again and choosed the best value according to the min fitness point.

4. Visualization:

- Create a grid where each row represents a robot and each column represents a task.
- Use color intensity to indicate task duration, with annotations for significant durations.
- Annotate each row with the robot's efficiency and each column with the task's priority.
- Check the example code section where the code is partially done.
- Annotate each cell with task priority and duration

to implement this we have used matplotlib

```
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))  #size of the figure in inches.
    cmap = mcolors.LinearSegmentedColormap.from_list("", ["white", "blue"])  # Custom colormap
    #white for low values (no task assigned) and blue for high values (task assigned).

    # 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)):

            # Determine text color based on task duration and assignment
            if grid[j, i] > 0 and task_durations[i] >= 5:
                text_color = 'white'  # White text for assigned tasks with duration >= 5
            else:
                text_color = 'black'  # Black text for unassigned tasks or tasks with duration < 5

            # Construct annotation text
            text = f'Priority:{task_priorities[i]}\n\n{task_durations[i]}Hours'
```

```
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()
```

specifically these line of code put the priority and duration value inside the grid and text color according to them.