Faculty of Engineering and Technology
Electrical and Computer Engineering Department

Artificial Intelligence
ENCS3340

Project #1
Optimizing Job Shop Scheduling in a Manufacturing Plant using Genetic Algorithm

Prepared by:
Omar Hussain 1212739
Malak Moqbel 1210608

Instructor: Ismail Khater

Section: 2
Date: 20/5/2024

Table of Contents

Table of figures:

Genetic Algorithm

Genetic Algorithms (GAs) solve problems by mimicking natural selection and genetics, evolving a population of possible solutions. The state space is the collection of all potential solutions, which the algorithm explores by selecting, combining, and mutating individuals to find the best answers. The fitness function measures the quality of each solution, with higher scores indicating better solutions. The selection operator picks the best solutions to create the next generation, while the crossover operator combines parts of two parent solutions to produce new offspring. The mutation operator introduces random changes to maintain diversity and avoid local optima. Ultimately, the best result is the solution with the highest fitness score, representing the optimal outcome and demonstrating the algorithm's ability to effectively solve complex problems.

Problem formulation:

This project develops a genetic algorithm to optimize job shop scheduling in a manufacturing plant with various machines, such as cutting, drilling, and assembly stations. Each product requires a specific sequence of operations on these machines. The system takes input as a list of jobs, each defined by a sequence of operations, specifying the machine and processing time for each task, and the number of available machines. The goal is to determine the optimal sequence and timing for each product to minimize overall production time or maximize throughput, considering machine capacities and job dependencies. The output is a schedule showing the start and end times for each process on each machine, visualized using a Gantt Chart.

1. Code

1.1 get_int_input

```python

# Function to get integer input from the user with optional maximum value validation
def get_int_input(prompt, max_val=None):
    while True:
        try:
            value = int(input(prompt))
            if value > 0 and (max_val is None or value <= max_val):
                return value
            elif max_val is not None and value > max_val:
                print(f"Please enter a value less than or equal to {max_val}.")
            else:
                print("Please enter a positive integer greater than 0.")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")
```

Figure 1 get_int_input

Function prompts the user to enter a positive whole number, continuing to ask until a valid input is provided. It uses a loop (`while True`) to repeatedly request input, attempting to convert it to an integer within the loop. If the number is greater than 0 and, if specified, less than or equal to a maximum value (`max_val`), it returns the number. If the number exceeds `max_val`, it instructs the user to enter a smaller number, and if it is not positive, it asks for a positive number. If the input cannot be converted to an integer, such as when letters are entered, it displays an error message and prompts the user again.

1.2 get_user_input

```python
# Function to gather user input for jobs, machines, and operations
def get_user_input():
    operations = {}
    num_jobs = get_int_input("Enter the number of jobs: ")
    num_machines = get_int_input("Enter the number of available machines: ")
    total_operations = 0
    for job in range(1, num_jobs + 1):
        job_operations = []
        num_operations = get_int_input(f"Enter the number of operations for Job {job}: ")
        total_operations += num_operations
        for op in range(1, num_operations + 1):
            machine = get_int_input(f"Enter the machine number for Job {job}, Operation {op} (1-{num_machines}): ", max_val=num_machines)
            time = get_int_input(f"Enter time required for Job {job}, Operation {op}: ")
            job_operations.append((f'M{machine}', f'Job{job}', f'Operation{op}', time))
        operations[f'Job{job}'] = job_operations
    return operations, num_machines, total_operations
```

Figure 2 get_user_input

Function gathers user inputs for job scheduling, starting by asking for the total number of jobs and available machines. It then loops through each job, prompting the user for the number of operations for that job. For each operation, it asks for the machine number and the required time, ensuring inputs are valid. The function then returns this dictionary along with the total number of machines and operations.

1.3 create_random_chromosomes

```python
# Function to create a random chromosome (a possible solution)
def create_random_chromosome(operations):
    job_keys = list(operations.keys())
    random.shuffle(job_keys)  # Shuffle job keys to randomize the order of jobs
    chromosome = []
    scheduled_operations = [(job, 0) for job in job_keys]  # Initialize each job with its first operation
    while scheduled_operations:
        for job_index in range(len(scheduled_operations)):
            job, op_index = scheduled_operations[job_index]
            if op_index < len(operations[job]):
                chromosome.append(operations[job][op_index])  # Append the operation to the chromosome
                scheduled_operations[job_index] = (job, op_index + 1)  # Move to the next operation of the job
        scheduled_operations = [(job, op_index) for job, op_index in scheduled_operations if op_index < len(operations[job])]
    return chromosome
```

Figure 3 create_random_chromosome

The `create_random_chromosome` function generates a random sequence of job operations for a genetic algorithm in job shop scheduling. It starts by shuffling the list of jobs, then iteratively adds each job's operations to the chromosome list in a random order while ensuring all operations for each job are included. The function returns this list of operations, where each operation specifies the machine, job, operation number, and required time. This randomized sequence serves as a possible solution for the genetic algorithm to optimize.

Example: [('M1', 'Job1', 'Operation1', 10), ('M2', 'Job2', 'Operation1', 7), ('M2', 'Job1', 'Operation2', 5),('M3', 'Job2', 'Operation2', 15), ('M1', 'Job2', 'Operation3', 8)]

1.4 Calculate_fitness

```python
# Function to calculate the makespan (total time) of a chromosome
def calculate_makespan(chromosome):
    machine_end_times = {}  # Track the end time for each machine
    job_end_times = {}  # Track the end time for each job
    for machine, job, operation, time in chromosome:
        start_time = max(machine_end_times.get(machine, 0), job_end_times.get(job, 0))  # Get the earliest possible start time
        end_time = start_time + time  # Calculate end time for the operation
        machine_end_times[machine] = end_time  # Update machine end time
        job_end_times[job] = end_time  # Update job end time
    return max(machine_end_times.values())  # Makespan is the latest end time among all machines

# Function to calculate the fitness of a chromosome (inverse of makespan)
def calculate_fitness(chromosome):
    return 1 / calculate_makespan(chromosome)

```

Figure 4 calculate_fitness

calculate_makespan:
This function calculates the makespan, which is the total time required to complete all operations in a given sequence (chromosome).
It uses two dictionaries, machine_end_times and job_end_times, to keep track of the end times for each machine and each job, respectively.
For each operation in the chromosome, it determines the earliest possible start time by taking the maximum of the machine's end time and the job's end time.
It then calculates the end time for the operation and updates the end times for the respective machine and job.
Finally, the function returns the maximum value from machine_end_times, representing the makespan, as it is the latest end time among all machines.
calculate_fitness:
This function calculates the fitness of a chromosome, which is the inverse of the makespan. A lower makespan indicates a better solution, so taking the inverse ensures that higher fitness values correspond to better solutions.

1.5 create_random_chromosomes

```python
# Function to select the two best chromosomes from the population
def select_two_best(population):
    sorted_population = sorted(population, key=lambda x: calculate_fitness(x), reverse=True)
    return sorted_population[:2]  # Return the two chromosomes with the highest fitness
```

Figure 5 create_random_chromosomes

Function selects the top two chromosomes from a given population based on their fitness. It sorts the population in descending order of fitness using the "calculate_fitness" function as the sorting key, ensuring that chromosomes with higher fitness values are prioritized. The function then returns the first two chromosomes from this sorted list, representing the two best solutions in the population.

1.6 order_crossover

```python
# Function to perform order crossover between two parent chromosomes
def order_crossover(parent1, parent2):
    size = len(parent1)
    point1, point2 = sorted(random.sample(range(size), 2))  # Select two crossover points
    child1_part = parent1[point1:point2]  # Get the part of parent1 between the crossover points
    child2_part = parent2[point1:point2]  # Get the part of parent2 between the crossover points

    def fill_child(part, parent):
        child = part[:]
        parent_index = point2
        while len(child) < size:
            operation = parent[parent_index % size]
            if operation not in child:  # Ensure no duplicates in child
                child.append(operation)
            parent_index += 1
        return child

    child1 = fill_child(child1_part, parent2)
    child2 = fill_child(child2_part, parent1)
    return child1, child2
```

Figure 6 order_crossover

Determine Size and Points:
The function starts by determining the size of the parent chromosomes.It then randomly selects two crossover points within the chromosome length.
Extract Segments:
From each parent chromosome, it extracts the segments between the two crossover points.
Fill Child Function:
The fill_child function helps in constructing the complete offspring by ensuring no duplicate operations are present.
It starts with the segment extracted from one parent and then fills in the remaining operations from the other parent, maintaining the order and ensuring uniqueness.
Construct Children:
The segments from the parents are used to create the initial parts of the children.
The fill_child function is called to fill in the rest of the children's chromosomes.
Return Children:
The function returns two children that are a mix of the two parents, incorporating parts of both parents' genes.

Example :

Parent 1:
[('M1', 'Job1', 'Operation1', 10), ('M2', 'Job1', 'Operation2', 5), ('M3', 'Job2', 'Operation1', 7), ('M4', 'Job2', 'Operation2', 8), ('M5', 'Job3', 'Operation1', 6)]
Parent 2:
[('M3', 'Job2', 'Operation1', 7), ('M1', 'Job1', 'Operation1', 10), ('M5', 'Job3', 'Operation1', 6), ('M4', 'Job2', 'Operation2', 8), ('M2', 'Job1', 'Operation2', 5)]

Child 1:
[('M2', 'Job1', 'Operation2', 5), ('M3', 'Job2', 'Operation1', 7), ('M1', 'Job1', 'Operation1', 10), ('M5', 'Job3', 'Operation1', 6), ('M4', 'Job2', 'Operation2', 8)]
Child 2:
[('M1', 'Job1', 'Operation1', 10), ('M5', 'Job3', 'Operation1', 6), ('M2', 'Job1', 'Operation2', 5), ('M3', 'Job2', 'Operation1', 7), ('M4', 'Job2', 'Operation2', 8)]

1.7 is_valid_chromosome

```python
# Function to check if a chromosome is valid (operations are in the correct order)
def is_valid_chromosome(chromosome, operations):
    last_indices = {}
    for operation in chromosome:
        machine, job, op, time = operation
        op_index = operations[job].index(operation)
        if job in last_indices:
            if op_index <= last_indices[job]:  # Check if the operation is in the correct order
                return False
        last_indices[job] = op_index  # Update the last operation index for the job
    return True
```

Figure 7 is_valid_chromosome

function checks if a chromosome is valid by ensuring that the operations for each job are in the correct order. It does this by iterating through the operations in the chromosome and comparing the indices of each operation to ensure they follow the defined sequence.

1.8 mutate

```python
# Function to mutate a chromosome
def mutate(chromosome, operations, mutation_rate):
    if random.random() > mutation_rate:  # Only mutate with a certain probability
        return chromosome, chromosome

    machine_ops = {}
    for op in chromosome:
        machine = op[0]
        if machine not in machine_ops:
            machine_ops[machine] = []
        machine_ops[machine].append(op)

    eligible_machines = [m for m in machine_ops if len(machine_ops[m]) > 1]  # Find machines with more than one operation
    if not eligible_machines:
        return chromosome, chromosome

    machine = random.choice(eligible_machines)  # Select a random eligible machine
    op1, op2 = random.sample(machine_ops[machine], 2)  # Select two random operations from the machine

    new_chromosome = []
    for op in chromosome:
        if op == op1:
            new_chromosome.append(op2)  # Swap operations
        elif op == op2:
            new_chromosome.append(op1)
        else:
            new_chromosome.append(op)

    if is_valid_chromosome(new_chromosome, operations):  # Ensure the mutated chromosome is valid
        return chromosome, new_chromosome
    return chromosome, chromosome
```

Figure 8 mutate

If a randomly generated number is higher than the mutation_rate, the function returns the original chromosome without any changes.Create a dictionary to track operations performed by each machine.Populate this dictionary by iterating over the operations in the chromosome.

Identify machines that have more than one operation. These machines are eligible for mutation. If no machines are eligible, return the original chromosome.Randomly select one eligible machine.

Randomly pick two operations from this machine.
Create a new chromosome by swapping the selected operations.
Check if the new chromosome maintains the correct order of operations.
If valid, return both the original and new chromosome.
If not valid, return the original chromosome twice.

Example :
Original Chromosome:
[**('M1', 'Job1', 'Operation1', 10)**, ('M2', 'Job2', 'Operation1', 7), ('M3', 'Job1', 'Operation2', 5), ('M2', 'Job3', 'Operation1', 8), **('M1', 'Job2', 'Operation2', 6)**]

Mutated Chromosome:
[('M1', 'Job2', 'Operation2', 6), ('M2', 'Job2', 'Operation1', 7), ('M3', 'Job1', 'Operation2', 5), ('M2', 'Job3', 'Operation1', 8), **('M1', 'Job1', 'Operation1', 10)**]

1.9 genetic_algorithm_loop

```python
# Function for the genetic algorithm loop
def genetic_algorithm_loop(initial_population, operations, num_crossovers, mutation_rate):
    population = initial_population
    for _ in range(num_crossovers):
        best_two = select_two_best(population)  # Select the two best chromosomes
        new_chromosome1, new_chromosome2 = order_crossover(best_two[0], best_two[1])  # Perform crossover
        before_mutation1, new_chromosome1 = mutate(new_chromosome1, operations, mutation_rate)  # Mutate the first child
        before_mutation2, new_chromosome2 = mutate(new_chromosome2, operations, mutation_rate)  # Mutate the second child

        if is_valid_chromosome(new_chromosome1, operations):  # Check if the first child is valid
            population.append(new_chromosome1)

        if is_valid_chromosome(new_chromosome2, operations):  # Check if the second child is valid
            population.append(new_chromosome2)

        population = select_two_best(population)  # Keep only the two best chromosomes in the population

    best_chromosome = select_two_best(population)[0]  # Return the best chromosome after all iterations
    return best_chromosome
```

Figure 9 genetic_algorithm_loop

Function refines a population of chromosomes to discover the optimal solution for job scheduling. It begins with an initial population and repeatedly selects the two best chromosomes, performing a crossover to produce two new offspring. These offspring are then subject to mutation based on a specified probability. If the mutated offspring are valid, they are added to the population. To maintain a fixed population size, only the top two chromosomes are retained after each iteration. This loop continues for a set number of crossovers, ultimately returning the best chromosome identified through the process.

1.10 calculate_schedule

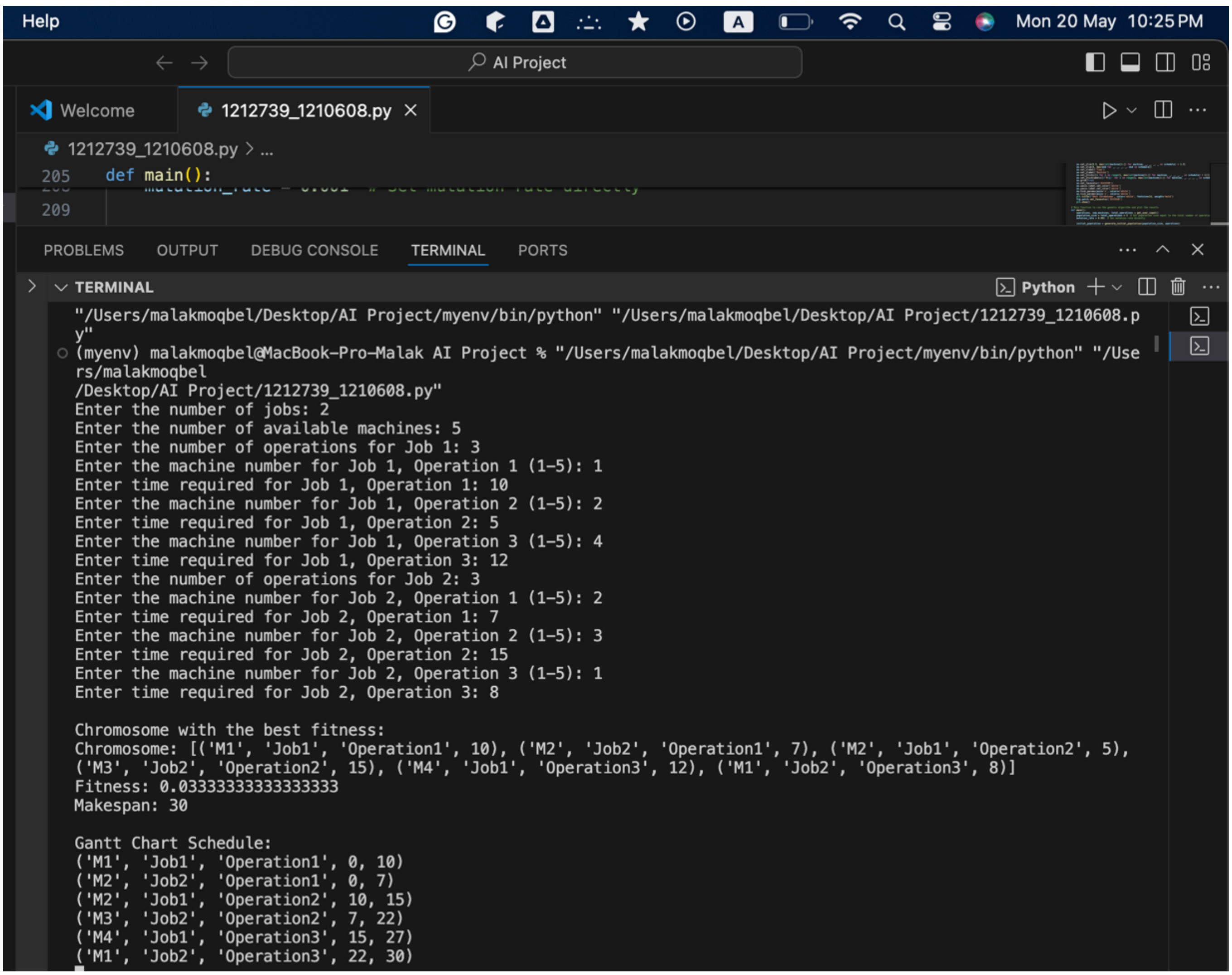
```python
# Function to calculate the schedule from a chromosome
def calculate_schedule(chromosome):
    machine_end_times = {}
    job_end_times = {}
    schedule = []
    for machine, job, operation, time in chromosome:
        start_time = max(machine_end_times.get(machine, 0), job_end_times.get(job, 0))  # Get the earliest possible start time
        end_time = start_time + time  # Calculate end time for the operation
        schedule.append((machine, job, operation, start_time, end_time))  # Append the schedule entry
        machine_end_times[machine] = end_time  # Update machine end time
        job_end_times[job] = end_time  # Update job end time
    return schedule

```

Figure 10 calculate_schedule

Function creates a schedule from a given chromosome by calculating the start and end times for each operation. It uses dictionaries to keep track of the end times for each machine and job. For each operation in the chromosome, it finds the earliest possible start time based on the current end times of the relevant machine and job, then calculates the end time. The function updates the end times in the dictionaries and adds the operation's details to the schedule. The final output is a detailed schedule showing the start and finish times for each operation on each machine.

2. Results

2.1 Test Case 1



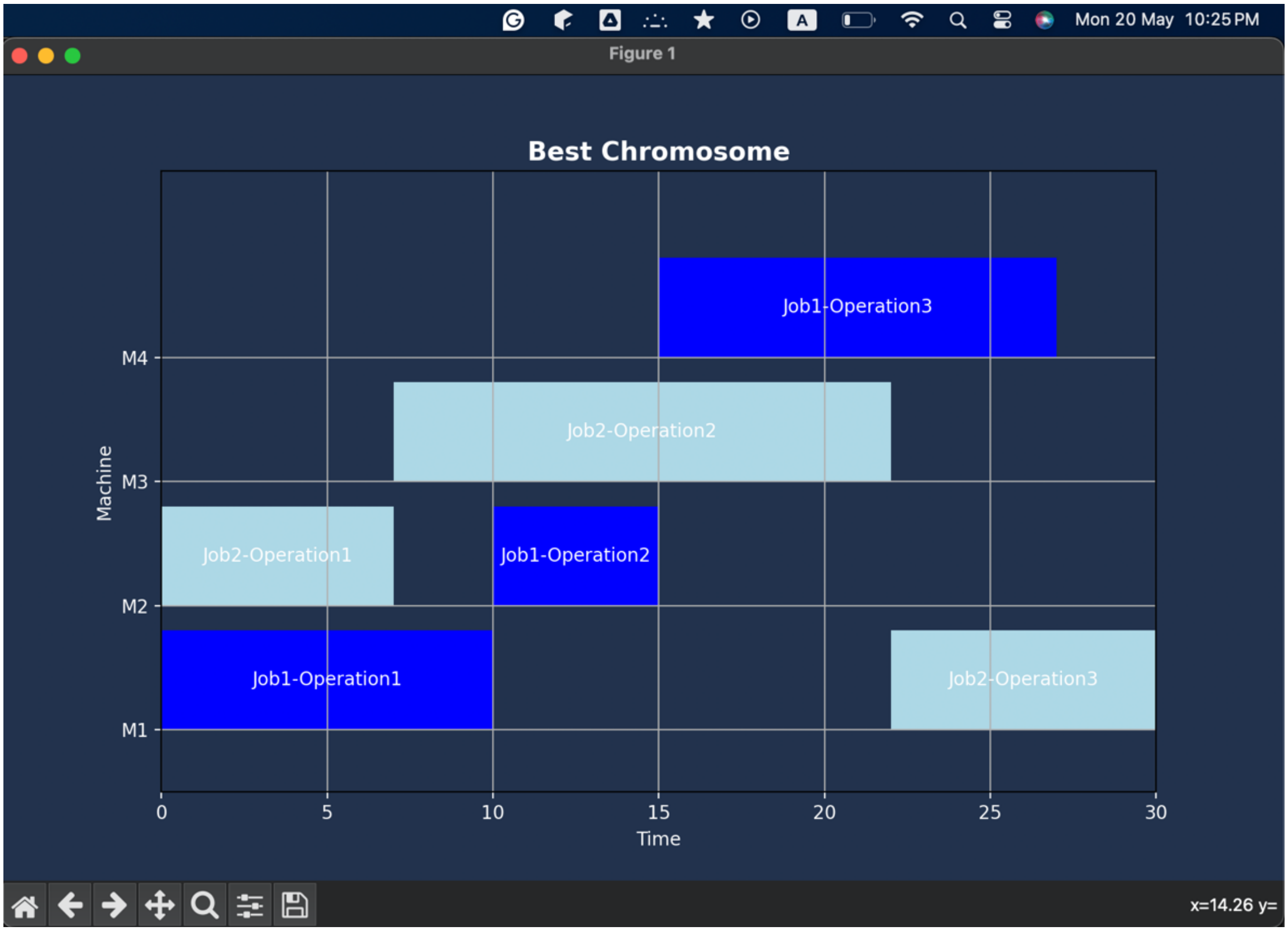


Figure 11 Test Case 1

2.2 Test Case 2

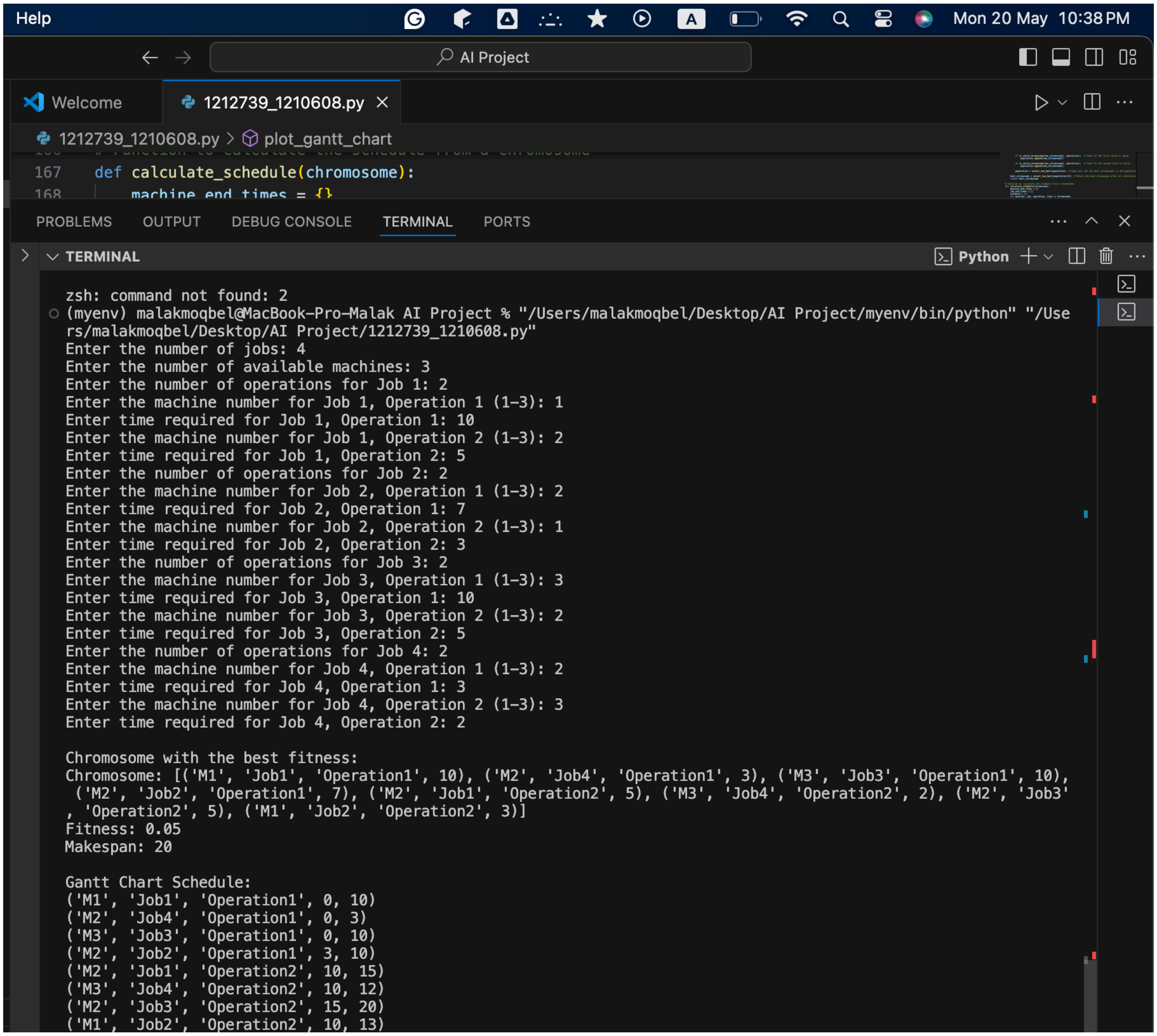


Figure 12 Test Case 2

The resulting Gantt chart successfully demonstrates that each machine works on only one job at a time and that all operations are executed sequentially. Each job's operations are carried out in the correct order, ensuring that, for example, Job1's second operation starts only after its first operation is completed. Machines handle one job at a time without overlap, as seen with Machine M1 completing Job1's first operation before starting Job2's second operation.

This outcome confirms that the genetic algorithm has performed as intended, producing an optimal schedule that minimizes the overall completion time while maintaining proper job and machine sequencing.