Faculty Of Engineering And Technology

Department of Electrical & Computer Engineering

Second Semester, 2023/2024

Artificial Intelligence

ENCS3340

Project 1:

Optimizing Job Shop Scheduling in a Manufacturing Plant using Genetic Algorithm

Name : Donia Alshiakh

ID : 1210517

Name : Shahd Yahya

ID : 1210249

Section : 2

Instructor: Dr. Ismail Khater

❖ Design philosophy and Result Analyzer:

❖ Important library :

```
import random #  'random' module, which provides functions to generate random
numbers.
from collections import defaultdict # Importing defaultdict from the
collections module, which provides a dictionary-like object that initializes
a default value if the key does not exist.
```

The random module provides functions to generate random numbers and perform random operations, which can be useful in various applications like simulations, games, and randomized algorithms.

Key Functions , we use:

- **random.randint(a, b)**: Returns a random integer between a and b (inclusive).
- **random.sample(population, k)**: Returns a list of k unique elements chosen from the population sequence.

The **defaultdict** is a subclass of the built-in dict class. It overrides one method and adds one writable instance variable. The main advantage of using **defaultdict** is that it provides a default value for the dictionary key that does not exist.

Key Functions, we use:

- **default_factory attribute**: A function that provides the default value for a nonexistent key. For example, **defaultdict(int)** creates a dictionary where missing keys return 0 by default.

❖ Classes :

```
class Job: # this to create object from job , each object jub must has two
valus index and taske
    def __init__(self, index, tasks):
        self.index = index
        self.tasks = tasks

class Task: # this to create object from Task , each object jub must has
seven valus index and taske
    def __init__(self, task, job_index, duration, task_order,
started_time=None, ended_time=None):
        # initialize the Task object with attributes:
        self.task = task  # reprecent machine on which the task will run
        self.job_index = job_index  # this  index of the job to which this
task belongs
```

```
        self.duration = duration
        self.task_order = task_order  # The order of the task in the job
sequence

        self.started_time = started_time  # (initialized as None)
        self.ended_time = ended_time  #  (initialized as None)
        self.waited_time = 0  # the amount of time the task has waited to be
processed (initialized to 0)
class Machine:
    def __init__(self, tasks=None, available=True):
        # to initialize the Machine object with attributes:
        if tasks is None: # If no tasks are provided,
            tasks = []  #initialize an empty list of tasks
        self.tasks = tasks  # The list of tasks assigned to this machine
        self.available = available  # The availability status of the machine
(initialized to True)

# Function to red jobs from input file (jobs.txt)
```

- **Class Job:**

This class is used to create Job objects. Each Job object must have two attributes: an index and a list of tasks.

- **__init__(self, index, tasks)**: This is the initializer method that sets up the Job object with the given index and tasks.
  1. **index**: The unique identifier for the job.
  2. **tasks**: A list of Task objects associated with this job.
- **Class Task:**

This class is used to create Task objects. Each Task object must have seven attributes, although some can be initialized as None.

- **__init__(self, task, job_index, duration, task_order, started_time=None, ended_time=None)**: This is the initializer method that sets up the Task object with the given attributes.
  1. **task**: Represents the machine on which the task will run.
  2. **job_index**: The index of the job to which this task belongs.
  3. **duration**: The time duration needed to complete the task.
  4. **task_order**: The order of the task in the job sequence.
  5. **started_time**: The start time of the task (initialized as None).
  6. **ended_time**: The end time of the task (initialized as None).
  7. **waited_time**: The amount of time the task has waited to be processed (initialized to 0).

- **Class Machine**

  This class is used to create Machine objects. Each Machine object can have a list of tasks and an availability status.

- **__init__(self, tasks=None, available=True)**: This is the initializer method that sets up the Machine object with the given attributes.
  1. **tasks**: A list of tasks assigned to this machine. If no tasks are provided, it initializes an empty list of tasks.
  2. **available**: The availability status of the machine (initialized to True).

❖ Function read from input file :

```
def read_file (name_file):
    jobs = [] # List of jobs
    with open(name_file, 'r') as file: # to pen the file for reading
        for line in file:# repeat over each line in the file

            if line.strip():# this check if the line is not empty
                print(line.strip()) # print the line without leading or
trailing whitespaces
                job_operate , tasks_str = line.split(':')# to split the line
into job index and task descriptions
                job_operate  = int(job_operate .split('_')[1])# Extract the
job index and convert it to an integer
                tasks = []  # to start an empty list to store tasks
                task_descriptions = tasks_str.strip().split(' -> ')# split
the task descriptions into individual tasks
                order = 1 # initialize the order of tasks from 1
                for task_desc in task_descriptions: #for loop Iterate over
each task description
                    machine, duration = task_desc.strip().split('[')# this
split the task description into machine and duration
                    machine = machine.strip()[1:] # this extract the machine
name and remove leading whitespace
                    duration = duration.strip(']') # this remove trailing ']'
from duration and convert it to an integer
                    tasks.append(Task(machine, job_operate, int(duration),
order))# to create a Task object and append it to the tasks list
                    order += 1 # to increment the order for the next task
                jobs.append(Job(job_operate, tasks)) # to create a Job object
with the extracted job index and tasks, then append it to the jobs list

    return jobs # this to return the list of jobs read from the file
```

The **read_file** function reads job and task information from a file. It initializes an empty list to store jobs, opens the specified file, and processes each line. For each non-empty line, it extracts the job index and task descriptions, creating a list of tasks for each job. Each task description includes the machine name and duration. The function returns a list of jobs, each containing its index and associated tasks. This allows for easy access to job and task data from the file.

This function acts like a translator, taking the information from the file in a specific format and converting it into a more structured representation (a list of jobs with their tasks) that your program can use more easily.

❖ function initialize population

```
# This function initialize population randomly by uses library from imprt
rando
def Init_Population(pop_size, num_jobs): # pop_size : population size
    # this to reate an initial population of chromosomes, each representing a
permutation of job indices
    return [[x + 1 for x in random.sample(range(num_jobs), num_jobs)] for _
in range(pop_size)]
 # ecch first job start from 1 not zero
```

- **pop_size**: This parameter specifies the size of the population, i.e., the number of chromosomes to generate.
- **num_jobs**: This parameter indicates the number of jobs for which permutations need to be created.

The function returns a list of lists, where each inner list represents a chromosome (a permutation of job indices). Each chromosome consists of integers representing job indices. The job indices are generated randomly using **random.sample(range(num_jobs), num_jobs)**, ensuring that each chromosome contains a unique permutation of job indices.

Example: Suppose we have 3 jobs and want to generate a population of 10 chromosomes.

Output :

[[2, 3, 1], [3, 1, 2], [1, 3, 2], [2, 1, 3], [3, 2, 1], [1, 2, 3], [1, 2, 3], [3, 2, 1], [2, 1, 3], [2, 1, 3]]

❖ Function of Create Scheduling :

```
❖  def Main_Schedule(jobs, pop, num_generations, mutate_rate,
   competition_size): #pop = population
       for g in range(num_generations): #g :generations
           computation_pop = []  # List to store the evaluated population
           for chro in pop: #chro : chromosome
               # to evaluate each chromosome using the scheduling function
               fitness, _ = Scheduling(chro, jobs) # best fitness (lower
   total time)
               computation_pop.append((chro, fitness))
           parents = SelectParents(computation_pop,competition_size)
           # this to select parents for the next generation
```

```
        new_pop = []  # List to store the new population
        for i in range(0, len(parents), 2):
            p1, p2 = parents[i], parents[i + 1]  # Select two parents # p1 :
perent1 ,, p2 : perent 2
            # Perform crossover to produce children
            ch1, ch2 = Crossover(p1, p2) # ch1 :child1  # ch2 :child2
            # Mutate the children
            ch1 = Mutation(ch1, mutate_rate)  # ch1 :child1
            ch2 = Mutation(ch2, mutate_rate)# ch2 :child2
            new_pop.extend([ch1, ch2])  # to add the children to the new
population
        pop = new_pop  # to update the population

    # to find the best chromosome with better fitness (lower total time)  in
the computation_pop
    best_chromosome = min(computation_pop, key=lambda x: x[1])[0]
    # to get the final schedule using the best chromosome
    _, final_schedule = Scheduling(best_chromosome, jobs)

    print("Best Chromosome:", best_chromosome)  # Print the best chromosome

    return final_schedule  # Return the final schedule
```

This function, **Main_Schedule**, is the main driver for the scheduling process using a genetic algorithm.

- **job:** This parameter represents the list of jobs that need to be scheduled.
- **pop**: This parameter represents the initial population of chromosomes.
- **num_generations**: This parameter specifies the number of generations (iterations) the genetic algorithm will run.
- **mutate_rate**: This parameter represents the probability of mutation for each chromosome during the evolution process.
- **competition_size**: This parameter specifies the size of the tournament for selecting parents during the evolution process.
  - **for g in range(num_generations):**

This loop iterates over each generation in the specified range of num_generations.

  - computation_pop = [] # List to store the evaluated population

A list named computation_pop is initialized to store the evaluated population, which will contain tuples of chromosomes and their corresponding fitness values.

  - **for chroromosome in pop:**
  - **fitness, _ = Scheduling(chro, jobs)**

Within each generation, the fitness of each chromosome in the population is evaluated using the

Scheduling function.

❖ Function of Scheduling:

```
# This function take bast chromosomes with best fitness (lower total time) to
order tasks on machines and evaluated fitness for each chromosome
def Scheduling(chromosome, jobs):
    current_time = 0  # initialize the current time
    job_position = [0] * len(jobs)  # To track the current task index for
each job
    temp_machine = defaultdict(Machine)  # Temporary storage for machines and
their tasks
    temp_machine_ordered = defaultdict(Machine)  # this store ordered tasks
based on the chromosome
    machines = defaultdict(Machine)  # final machines and their tasks
    total_waiting_time = 0  # this total waiting time across all tasks
    for job in jobs: # in each loop take one job from jobs
        for task in job.tasks:
            task.waited_time = 0 # initialize the waiting time for each task
to 0
    # Populate temp_machine with tasks
    for job in jobs:
        for task in job.tasks:
            if task.task not in temp_machine:
                temp_machine[task.task] = Machine()
            temp_machine[task.task].tasks.append(task)
    while True:
        # check if tasks on machines are completed at the current time
        for machine in machines:
            for task in machines[machine].tasks:
                if task.ended_time == current_time:
                    if job_position[task.job_index - 1] <
len(jobs[task.job_index - 1].tasks):
                        job_position[task.job_index - 1] += 1  # Move to the
next task in the job
                    machines[machine].available = True  # Mark the machine as
available
        # to order tasks on temporary machines based on the chromosome
        for machine in temp_machine:
            for i in chromosome:
                for task in temp_machine[machine].tasks:
                    if i == task.job_index and task not in
temp_machine_ordered[task.task].tasks:
                        temp_machine_ordered[task.task].tasks.append(task)
        # assign tasks to machines if they are available and ready to start
        for machine_index in temp_machine_ordered:
            machine = machines[machine_index]
            if machine.available and
temp_machine_ordered[machine_index].tasks:
                task = temp_machine_ordered[machine_index].tasks[0]
                if task.task_order - 1 == job_position[task.job_index - 1]:
# check if the task is the next in order
                    task.started_time = current_time  # set the start time
for the task
                    task.ended_time = current_time + task.duration  # set the
end time for the task
```

```
                    machine.tasks.append(task)  # assign the task to the
machine
                    temp_machine_ordered[machine_index].tasks.remove(task)  #
remove task from ordered list
                    temp_machine[machine_index].tasks.remove(task)  # remove
task from temporary storage
                    machine.available = False  # mark the machine as busy

        current_time += 1  # increment the current time
        for machine in temp_machine:
            for task in temp_machine[machine].tasks:
                task.waited_time += 1  # increment the waiting time for each
task

        # This check if all jobs are complete
        complete = all(job_position[i] >= len(job.tasks) for i, job in
enumerate(jobs))
        if complete:
            break
    # calculate the total waiting time
    for machine in machines:
        for task in machines[machine].tasks:
            total_waiting_time += task.waited_time

    return current_time - 1, machines  # return the total time as fitness
and the final machine schedules
```

The function, Scheduling, orchestrates the allocation of tasks to machines based on a given chromosome representing the order in which jobs should be executed:

1. **Initialization:**

- current_time is set to 0 to indicate the starting time of the scheduling process.
- job_position is initialized as a list of zeros, where each element corresponds to the current task index for a particular job.
- Three defaultdicts are initialized:
    - temp_machine: Used as temporary storage for machines and their tasks.
    - temp_machine_ordered: Stores tasks ordered based on the chromosome.
    - machines: Represents the final assignment of tasks to machines.
- total_waiting_time is initialized to 0 to track the total waiting time across all tasks.

2. **Iterating Through Jobs:**

- The function loops through each job in the jobs list.
- For each task within the current job:
    - task.waited_time is initialized to 0, representing the initial waiting time of the task.

3. **Populating Temporary Machine Storage:**

    1. Another loop iterates through all jobs again.
    2. For each task within the current job:

- If the task's name (task.task) is not yet a key in temp_machine, a new Machine object is created and associated with that task name in temp_machine.

- The current task is appended to the tasks list of the corresponding machine in temp_machine. This temporarily groups tasks by machine based on their names (assuming they should be processed on the same machine).

4. **Main Scheduling Loop:**

   1. **A while True loop continues until all tasks are completed.**
   2. **Checking Machine Completion:**

   - The loop iterates through each machine in the machines dictionary.
   - For each task assigned to the current machine:

     - If task.ended_time is equal to current_time, it means the task has finished at the current time.

     - The job_position for the corresponding job is incremented by 1, indicating that the next task in that job is ready to be processed.
     - The machine.available flag is set to True, signifying that the machine is now free to handle new tasks.

   3. **Ordering Tasks Based on Chromosome:**

   - The loop iterates through each machine in temp_machine.
   - For each **i** (element) in the chromosome:

     - The loop iterates through each task in the current machine (from temp_machine).
     - If **i** (representing a job index) matches task.job_index (the job this task belongs to) and the task is not yet present in the tasks list of temp_machine_ordered for that machine:

       - The task is appended to the tasks list of the corresponding machine in temp_machine_ordered. This reorders tasks within each machine based on the order specified by the chromosome.

5. **Assigning Tasks to Machines:**

   1. The loop iterates through the indices (machine_index) in temp_machine_ordered.
   2. The corresponding machine object is retrieved from the machines dictionary.
   3. If the machine.available flag is True (meaning the machine is free) and there are tasks in temp_machine_ordered[machine_index].tasks:

- The first task from the tasks list of temp_machine_ordered for the current machine is retrieved.
- If the task.task_order - 1 is equal to job_position[task.job_index - 1]:
    - This check ensures that the task being assigned is the next one in order for its corresponding job (based on both chromosome order and job completion).
        - The task.started_time is set to current_time, marking the start time for this task.
        - The task.ended_time is calculated by adding the task's duration to the current_time, determining its completion time.
        - The task is appended to the tasks list of the machine in the

6. **Completion Check**:

- It verifies if all jobs are complete. If so, the loop breaks.

7. **Fitness Evaluation**:

- The function calculates the total time taken (fitness) for the schedule by subtracting 1 from the current_time. This represents the total time taken for all tasks to complete.
- It also computes the total waiting time across all tasks.

8. **Return**:

- The function returns the fitness (total time) and the final machine schedules.

➢ **Example:**

- Jobs:
    - Job_1: M1[5] -> M2[2] -> M3[2] -> M4[1]
    - Job_2: M3[5] -> M2[4] -> M4[3] -> M5[2]
    - Job_3: M1[9] -> M4[3] -> M5[2]
- Chromosome: [2, 3, 1]

Using the chromosome [2, 3, 1], the function will schedule the tasks according to this order and compute the total time taken and the final machine schedules.

➢ **The output is :**

M1 | 0[Job_3]9 9[Job_1]14 |

M2 | 0-----5[Job_2]9 9-----14[Job_1]16 |

M3 | 0[Job_2]5 5----------16[Job_1]18 |

M4 | 0---------9[Job_2]12 12[Job_3]15 15---18[Job_1]19 |

M5 | 0----------12[Job_2]14 14-15[Job_3]17 | **Total time** = 19

- Chromosome: [3, 2, 1] ---→ total time is 19
- Chromosome: [3, 1, 2] ---→ total time is 32
- Chromosome: [1, 3, 2] ---→ total time is 23
- Chromosome: [1, 2, 3] ---→ total time is 26

❖ **Complete code of Main_Scheduling :**

- **computation_pop.append((chro, fitness))**

The chromosome and its fitness value are then appended to the computation_pop list.

- **parents = SelectParents(computation_pop, competition_size)**

After evaluating the fitness of all chromosomes in the population, parents are selected for the next generation using tournament selection. The SelectParents function is called with competition_size specifying the size of the tournament.

❖ Function of select parents :

```
def SelectParents(pop, competition_size): #pop = population
    select_parents = []  # list to store selected parents
    for _ in range(len(pop)):
        # randomly select nominates for the tournament
        nominates = random.sample(pop, competition_size)
        # to choose the nominates with the best fitness
        winner = min(nominates, key=lambda x: x[1])
        select_parents.append(winner[0])  # Add the winner to the list of
selected parents
    return select_parents
```

- This line defines a function named SelectParents.
- The function takes two arguments:
  - pop: This is the population, which is likely a list of individuals (represented as sub-lists). In your example, each individual has a chromosome (the first sub-list) and a fitness value (the second element in the main list).
  - competition_size: This is the number of individuals to randomly select for the competition to choose a parent.

1. **Initializing Empty List:** **select_parents = []**

- This line creates an empty list named select_parents. This list will eventually store the selected parents chosen by the tournament selection process.

2. **Looping for Parents:** **for _ in range(len(pop)):**

- This for loop iterates len(pop) times. In your example, len(pop) is 10, so the loop will run 10 times, selecting two parents (one in each iteration).

3. **Randomly Selecting Nominees (Competitors):**

nominates = random.sample(pop, competition_size)

- Inside the loop, this line uses the random.sample function from the random module.
- random.sample(pop, competition_size) randomly selects competition_size individuals (sub-lists) from the pop list without replacement (meaning an individual can't be chosen twice in the same competition).
- This creates a new list named nominates that holds these randomly selected individuals for the current competition round.

4. **Finding the Fittest Nominee (Competitor):**
   winner = min(nominates, key=lambda x: x[1])

- This line finds the fittest (having the lowest fitness value) individual among the nominates.
- The min function is used, but with a custom key function (lambda x: x[1]).
  - The lambda expression defines an anonymous function that takes an individual (x) and returns its fitness value (x[1]).
- By using key=lambda x: x[1], the min function compares the fitness values of the individuals in the nominates list and returns the one with the **lowest** fitness value (the fittest one).

5. **Adding Winner to Selected Parents: select_parents.append(winner[0])**

- This line adds the first element (winner[0]) of the winner individual (which is the chromosome) to the select_parents list. This is the chromosome of the fittest individual who won the current competition round and is chosen as a parent.

➢ **Example:**

population = [[2, 3, 1], [3, 1, 2], [1, 3, 2], [2, 1, 3], [3, 2, 1], [1, 2, 3], [1, 2, 3], [3, 2, 1], [2, 1, 3], [2, 1, 3]]
competition size = 4
Assume two rounds of the selection process (will get two parents after two rounds):

- **Round 1:**

1. Randomly selected nominees**:** say the randomly chosen nominees are:
   - [[2, 3, 1], 3] (first individual)
   - [[1, 2, 3], 1] (second individual)
   - [[3, 2, 1], 2] (third individual)
   - [[2, 1, 3], 3] (fourth individual)

2. Finding fittest nominee: Among these nominees, the second individual [[1, 2, 3], 1] has the lowest fitness value (1).
3. Adding winner chromosome to parents: The first element (chromosome) of the winner ([1, 2, 3]) is appended to the select_parents list.

- **Round 2:** (Repeat the same steps as in Round1)

❖ **Complete code of Main_Scheduling :**

new_pop = [] # List to store the new population

A new list named new_pop is initialized to store the new population of chromosomes after crossover and mutation.

for i in range(0, len(parents), 2):

p1, p2 = parents[i], parents[i + 1]

ch1, ch2 = Crossover(p1, p2)

ch1 = Mutation(ch1, mutate_rate)

ch2 = Mutation(ch2, mutate_rate)

new_pop.extend([ch1, ch2])

For each pair of selected parents, **Crossover Function** is performed to generate two children. Then, each child undergoes **Mutation Function** based on the specified mutation rate. The resulting children are added to the new_pop list.

❖ **Function of Crossover** :

```
def Crossover(p1, p2): # p1 : perent1 ,, p2 : perent 2
    # Select a crossover point randomly
    crossover_point = random.randint(1, len(p1) - 1)
    # create children by combining genes from both parents at the crossover
point
    ch1 = p1[:crossover_point] + [gene for gene in p2 if gene not in
p1[:crossover_point]] # ch1 :child1
    ch2 = p2[:crossover_point] + [gene for gene in p1 if gene not in
p2[:crossover_point]] # ch2 :child2
    return ch1, ch2
```

The function implements single-point crossover, a genetic algorithm operation that combines genes from two parent chromosomes (individuals) to generate two new offspring (child) chromosomes.

- **Input:**
  - **p1**: The first parent chromosome (a list of genes).
  - **p2**: The second parent chromosome (another list of genes).
- **Output:**
  - **ch1**: The first child chromosome, created by combining genes from p1 and p2.
  - **ch2**: The second child chromosome, created using a similar combination strategy.
- **Step-by-Step:**

1. **Selecting a Crossover Point:**

- **crossover_point = random.randint(1, len(p1) - 1):**
  - This line generates a random integer (crossover_point) between 1 and the length of the first parent chromosome (p1) minus 1.
  - This value represents the position in the chromosomes where the gene exchange will occur.

➢ **Example:** Given p1 = [2, 3, 1], the valid crossover points range from 1 to 2 (excluding 0 and 3).

➢

2. **Creating the First Child Chromosome (ch1):**

- **ch1 = p1[:crossover_point] + [gene for gene in p2 if gene not in p1[:crossover_point]]:**
  - This line constructs the ch1 chromosome by combining genes from both parents.
  - The first part of ch1 is a slice of the p1 chromosome, containing genes from the beginning up to (but not including) the selected crossover_point.
  - The remaining genes for ch1 are selected from the p2 chromosome. However, only genes that are **not** already present in the first part of ch1 (the genes from p1) are added. This ensures that each child chromosome receives a unique set of genes.

➢ **Example:** Assume the randomly chosen crossover_point is 2.

  - The first part of ch1 is p1[:2], which is [2, 3].
  - To select the remaining genes for ch1, we need to find genes from p2 that are **not** in [2, 3].
  - Since p2 = [1, 3, 2], we find that only 1 is not in [2, 3], so we append it to ch1.
  - Therefore, ch1 becomes [2, 3, 1].

3. **Creating the Second Child Chromosome (ch2):**

- **ch2 = p2[:crossover_point] + [gene for gene in p1 if gene not in p2[:crossover_point]]:**
  - This line mirrors the process for creating ch1, but using p2 as the starting chromosome and p1 as the source for the remaining genes.

➢ **Example:** Continuing with the same crossover_point of 2, we create ch2:

- The first part of ch2 is p2[:2], which is [1, 3].
- To select the remaining genes for ch2, we need to find genes from p1 that are **not** in [1, 3].
- Since p1 = [2, 3, 1], we find that only 2 is not in [1, 3], so we append it to ch2. h
- Therefore, c2 becomes [1, 3, 2].

The Crossover function effectively combines genes from two parent chromosomes to generate two unique offspring chromosomes (ch1 and ch2). This process introduces new gene combinations that may lead to improved fitness in subsequent generations of the genetic algorithm.

❖ Function of Mutation :

```
❖ def Mutation (chro, mutate_rate): # chro : chromosome
      # this mutate the chromosome with a given mutation rate
      if random.random() < mutate_rate:
          in1, in2 = random.sample(range(len(chro)), 2)  #ind1 : index1
,, in2 : index2
          # to swap two genes in the chromosome
          chro[in1], chro[in2] = chro[in2], chro[in1]
      return chro
```

This function implements the mutation operation in a genetic algorithm. Mutation introduces random changes to a chromosome (individual) with a specified probability (mutation rate) to increase the diversity of the population and potentially lead to better solutions.

- **Input:**
  - **chro**: The chromosome to be mutated (a list of genes).
  - **mutate_rate**: The probability (between 0 and 1) that a mutation will occur.
- **Output:**
  - The mutated chromosome (chro).

- **Step-by-Step:**

1. **Determining Mutation:**

- **if random.random() < mutate_rate::**

  - This line generates a random number between 0 and 1.
  - If this random number is **less** than the **mutate_rate**, then a mutation will occur. Otherwise, the chromosome remains unchanged.

  ➢ **Example:** Given **mutate_rate = 0.001**, there is a 0.1% chance that a mutation will happen.

2. **Selecting Mutation Points:**

- **in1, in2 = random.sample(range(len(chro)), 2):**

  - If the mutation condition is met, this line randomly selects two **different** indices (in1 and in2) from the range of the chromosome's length (len(chro)).
  - These indices represent the positions of two genes that will be swapped in the mutation process.

  ➢ **Example:** Assume **chro = [2, 3, 1]** and a random sample generates **in1 = 0** and **in2 = 2**.

3. **Performing Mutation (Swapping Genes):**

- **chro[in1], chro[in2] = chro[in2], chro[in1]:**

  - This line swaps the genes at the selected indices (in1 and in2) within the chro chromosome.
  - This effectively introduces a random change to the order of genes in the chromosome.

  ➢ **Example:** After swapping, **chro** becomes **[1, 3, 2].**

4. Returning Mutated Chromosome:

- return chro:

  - The mutated chromosome (chro) is returned.

The **Mutation** function introduces random changes to a chromosome with a controlled probability, potentially leading to new gene combinations and increasing the diversity of the population. This can help the genetic algorithm explore a wider search space and potentially find better solutions.

❖ **Complete code of Main_Scheduling :**

- **pop = new_pop** # Update the population for the next generation

Once all children are generated, the `pop` variable is updated with the new population for the next generation.

- **best_chromosome = min(computation_pop, key=lambda x: x[1])[0]**
- **_, final_schedule = Scheduling(best_chromosome, jobs)**
- **print("Best Chromosome:", best_chromosome)**

After all generations, the best chromosome with the lowest fitness value (indicating the shortest scheduling time) is identified. The final schedule corresponding to the best chromosome is obtained using the `Scheduling` function. The best chromosome is printed, and the final schedule is returned.

- **return final_schedule** # Return the final schedule

Finally, the function returns the final schedule obtained from the best chromosome.

❖ Run the code:

```
# Print jobs from file
print("Contents of 'jobs.txt':")
jobs = read_file("jobs.txt")
# Get number of machines from user input
while True:
    num_machines = input("Enter the number of machines: ")
    if num_machines.lower() == "exit":
        exit()
    try:
        num_machines = int(num_machines)
        break
    except ValueError:
        print("Please enter a valid number or 'exit' to quit.")
# Check if the number of machines in the file matches the entered number
max_machine_num = max(int(task.task) for job in jobs for task in job.tasks)
if num_machines < max_machine_num:
    print(f"Number of machines provided ({num_machines}) is less than the
required ({max_machine_num})")
    print("Please enter a valid number of machines or type 'exit' to quit.")
    while True:
        num_machines = input("Enter the number of machines: ")
        if num_machines.lower() == "exit":
            exit()
        try:
            num_machines = int(num_machines)
```

```
            break
        except ValueError:
            print("Please enter a valid number or 'exit' to quit.")
# get population size based on the number of jobs
pop_size = len(jobs) * 10 if len(jobs) < 5 else len(jobs) * 100 # pop_size :
population size
population = Init_Population(pop_size, len(jobs))
num_generations = 100
mutate_rate = 0.001
competition_size = 4
schedule = Main_Schedule(jobs, population, num_generations, mutate_rate,
competition_size)
```

- **Read Jobs from File**:
  - The function **read_file("jobs.txt")** reads job descriptions from the file and stores them in a list called **jobs**.
  - The contents of the file are printed to the console.

- **Get Number of Machines**:
  - The user is prompted to enter the number of machines.
  - If the input is not a valid integer, the user is prompted again.
  - If the user types "**exit**", the program terminates.

- **Validate Number of Machines**:
  - The code checks if the entered number of machines is at least as many as required by the tasks described in the file.
  - If the entered number is less than required, the user is prompted to enter a valid number again or type **"exit"** to quit.

- **Initialize Population**:
  - The population size (**pop_size**) is determined based on the number of jobs. If there are fewer than 5 jobs, the population size is **10 * number of jobs. Otherwise, it is 100 * number of jobs.**
  - The initial population of chromosomes is created using the Init_Population function.

- **Set Parameters for Genetic Algorithm**:
  - **num_generations**: The number of generations for the genetic algorithm is set to 100.
  - **mutate_rate**: The mutation rate is set to 0.001.
  - **competition_size**: The size of the competition (tournament) for parent selection is set to 4.

- **Run Genetic Algorithm**:

  - The **Main_Schedule** function is called with the jobs, initial population, and other parameters to run the genetic algorithm.
  - The final schedule is stored in the variable **schedule**.

This code involves reading data, validating user input, initializing a population for a genetic algorithm, and then running the algorithm to optimize the scheduling of tasks on machines.

❖ The output prints similar to a Gantt Chart :

```
# Print the final schedule in the desired format
for machine_index, machine in schedule.items(): #loop through each machine
and its scheduled tasks in the final schedule
    # print the machine index in the desired format
    print(f"M{machine_index} | ", end="")
    current_time = 0  # Initialize the current time to 0
    # Loop through each task assigned to the current machine
    for task in machine.tasks:
        # Check if there is an idle period before the task starts
        if task.started_time > current_time:
            # Print the idle period as dashes
            print(f"{current_time}{'-' * (task.started_time -
current_time)}", end="")
        # Print the task's start and end times along with the job index
        print(f"{task.started_time}[Job_{task.job_index}]{task.ended_time}",
end=" ")
        current_time = task.ended_time  # Update the current time to the
task's end time
    print("|")
```

This code ensures the schedule is presented clearly, showing both tasks and idle periods on each machine.

- **Loop Through Each Machine**:

  - The code iterates through each machine and its scheduled tasks in the final schedule (**schedule**).

- **Print Machine Index**:

  - It prints the machine index in the format "**M{machine_index} |** " without a newline at the end.

- **Initialize Current Time**:

  - The variable **current_time** is initialized to 0. This variable keeps track of the current time as tasks are printed.

- **Loop Through Each Task**:

  - For each task assigned to the current machine:

    - If there's an idle period before the task starts, it prints the idle period as dashes.
    - It prints the task's start and end times along with the job index in the format **"{start_time}[Job_{job_index}]{end_time}".**
    - It updates **current_time** to the task's end time.

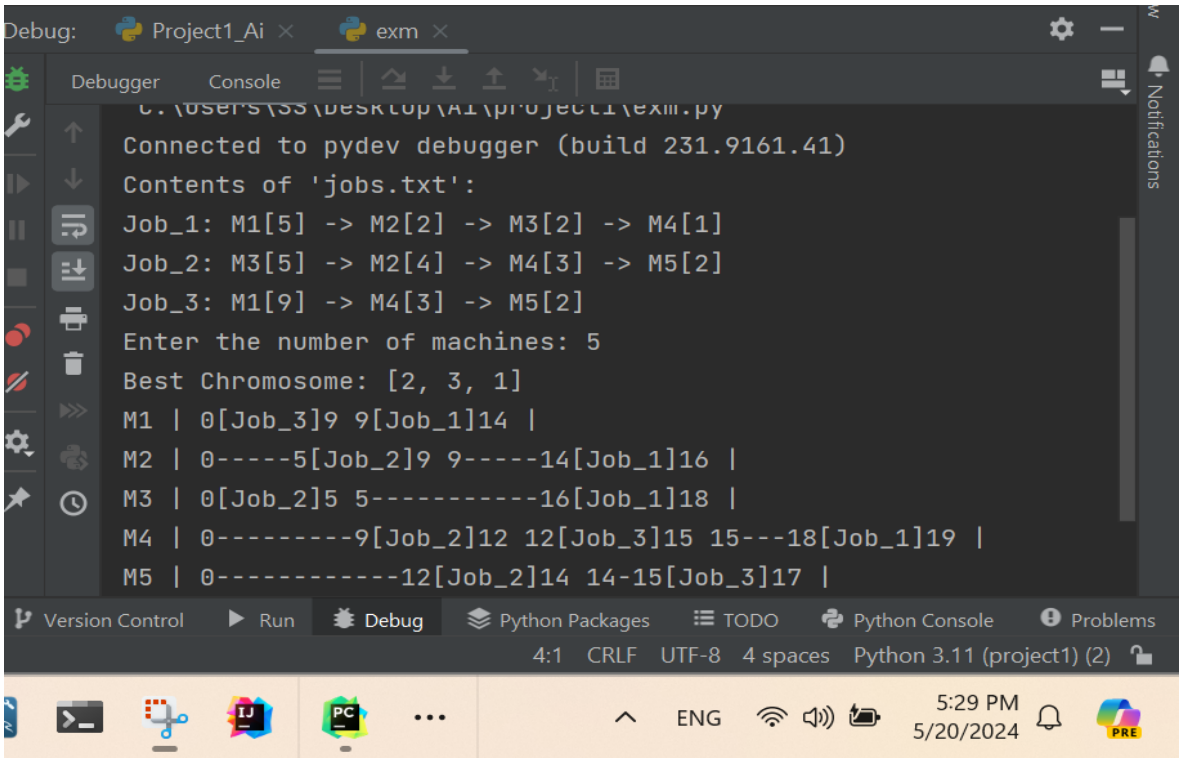
  - After printing all tasks for a machine, it prints a closing pipe character **(|)** and moves to the next line for the next machine.

❖ Examples 1 :

- **Input in "jobs.txt":**

```
Job_1: M1[5] -> M2[2] -> M3[2] -> M4[1]
Job_2: M3[5] -> M2[4] -> M4[3] -> M5[2]
Job_3: M1[9] -> M4[3] -> M5[2]
```

- **Output:**

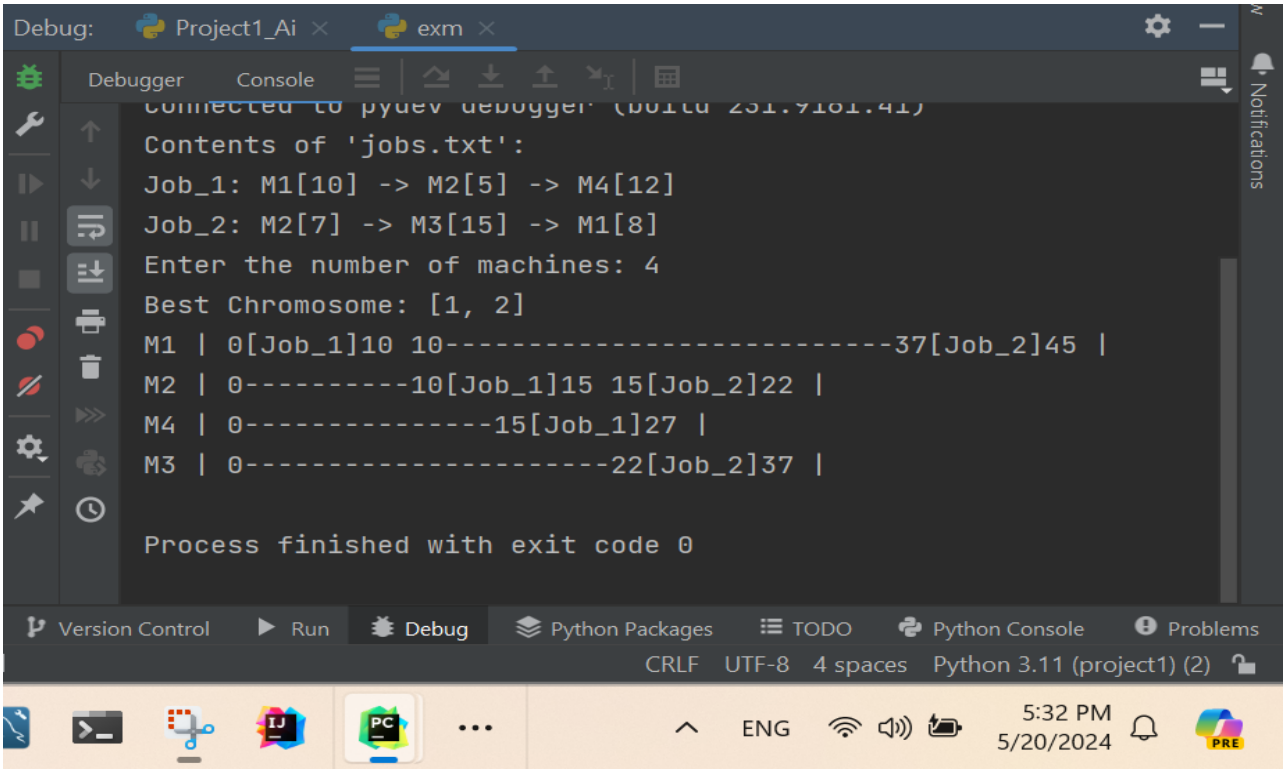


❖ Examples 2 :

- **Input in "jobs.txt":**

```
Job_1: M1[10] -> M2[5] -> M4[12]
Job_2: M2[7] -> M3[15] -> M1[8]
```

- **Output:**



- **Note :**If the number of machines entered by the user is less than the number required for each job.