Faculty of Engineering and Technology

Department of Electrical and Computer Engineering

ENCS3390

ARTIFICIAL INTELLIGENCE

FIRST PROJECT – GENETIC ALGORITHM

Prepared by: **Mayar Masalmeh** **1211246**

Lana Musaffer **1210455**

Instructor: **Dr. Yazan Abu Farha**

Section: **1**

Problem Formulation

Scheduling problem is an assignment problem, which can be defined as the assigning of available machines to the operations in such a manner that maximizes the profitability, flexibility, productivity, and performance of our manufacturing plant. Scheduling of operations is one of the most critical issues in the planning and managing of manufacturing processes.

The genetic algorithm belongs to the category of local search algorithm. We used this algorithm for optimizing job shop scheduling, it begins with a randomly generated population of potential solutions, each represented by a set of parameters called genes. These solutions are evaluated for their fitness in solving a given problem. The fittest individuals are selected to reproduce through crossover, where their genes are combined to create new solutions, and mutation, where some genes are randomly altered to introduce diversity. This process of selection, crossover, and mutation continues for several generations, with the population evolving towards better solutions until a satisfactory result is achieved.

1. State Representation:

In our genetic algorithm, each chromosome represents a potential scheduling solution using a state representation that includes the job number, machine number, and processing time. This is achieved through the encoding of operations as tuples containing job indices, machine identifiers, and corresponding processing times. The chromosome is manipulated through crossover and mutation operations to explore different scheduling configurations, ensuring each job is assigned to a machine for processing within the specified time constraints. The resulting scheduling solutions are evaluated and optimized iteratively to achieve efficient job processing.

2. Fitness Function:

To select the best-performing schedules in our program, we used makespan algorithm. The makespan is the total time required to complete all jobs in the schedule. It is calculated by simulating the execution of all jobs across different machines. The code first initializes the end times for all machines and jobs. As it iterates through each job and its operations, it tracks the start and end times for each operation, ensuring that operations on the same machine do not overlap and that each job's operations occur in sequence. The makespan is then determined as the maximum completion time across all jobs. The fitness function evaluates the quality of a given schedule by inversely relating it to the makespan. Specifically, it computes the fitness value as the scaled inverse of the makespan, using a predefined scaling factor. This means shorter makespans result in higher fitness values, encouraging the genetic algorithm to favor

schedules that complete all jobs more quickly. The scaling factor is used to convert the fitness value into a more manageable integer range, facilitating the selection and evolution processes within the genetic algorithm, and we set its value to 100.

In short, The newly created individuals (schedules) are indeed evaluated by calculating their makespan. The makespan serves as a direct measure of how well the schedule optimizes the completion of all jobs. The fitness value is calculated as the scaled inverse of the makespan. This fitness value is a numerical representation of the schedule's quality, with lower makespans resulting in higher fitness values.

3. Selection Criteria

The selection criteria used in the code is based on a proportional selection mechanism, specifically the roulette wheel selection method. This method involves assigning selection probabilities to each chromosome in the population, where the probability is directly proportional to the chromosome's fitness value. The fitness value is calculated as the scaled inverse of the makespan, meaning that schedules with shorter makespans have higher fitness values and thus higher probabilities of being selected. During the selection process, chromosomes are chosen randomly, but with a bias towards those with better performance (higher fitness). This ensures that high-quality solutions are more likely to contribute to the next generation, promoting the propagation of favorable traits and guiding the population towards more efficient schedules over successive iterations. This probabilistic selection process allows for both exploration and exploitation within the search space, maintaining genetic diversity while focusing on the most promising solutions. While our code doesn't explicitly outline the replacement strategy (whether all or only a subset of the population is replaced), it does follow a typical genetic algorithm approach where the new population is created by selecting parents based on their fitness, applying crossover and mutation, and then ensuring that the population remains valid.

4. Reproduce New Generations

- **Filtering for Next Generation:**

  Higher Probability for Better Chromosomes: Chromosomes with higher fitness values (better schedules with lower makespans) are more likely to be selected for the next generation. This filtering process ensures that the genetic algorithm incrementally improves the overall quality of the population over successive generations.

- **Mutation:**

  Mutation is used to produce perturbations on chromosomes in order to maintain the diversity of population. Two types of mutation are employed in our code, inversion mutation and insertion mutation:

- **Inversion mutation** serves to maintain the diversity in population, a segment of a chromosome corresponding to a specific job is randomly selected, and the order of operations within that segment is reversed. This process effectively reorders the sequence of operations within the selected job, potentially leading to the creation of novel schedules. By flipping the sequence of operations, inversion mutation facilitates exploration of alternative job sequences and contributes to the overall genetic diversity of the population.
- **Insertion mutation** randomly selects two operations within a chromosome and inserts the second operation before the first one. This mutation operation introduces a small perturbation to the chromosome's sequence, potentially altering the order of operations within or between jobs.
- **apply_mutation function**, it iterates through each chromosome in the population and applies mutation with a certain probability determined by the mutation rate parameter (mutation rate), which we set its value to 0.1 (each gene or operation in a chromosome has a 10% chance of being mutated during the genetic algorithm's evolution process). Inside the function, a mutation operator is chosen randomly from the available options, which in this case are inversion mutation and insertion mutation. Once selected, the chosen mutation operator is applied to a copy of the chromosome, producing a mutated version. This mutated chromosome is then added to the population of mutated individuals.

- **Cross-Over:**

  two parent schedules are selected from the population, and a random crossover point is chosen. A child schedule is then created by copying a segment from one parent up to the crossover point and filling in the remaining positions in the child with elements from the other parent that are not already present in the segment. This process allows for the exchange of genetic material between parent schedules, potentially producing offspring with characteristics inherited from both parents. The crossover rate parameter determines the likelihood of applying crossover during the evolution process, influencing the exploration of the search space for better solutions to the job scheduling problem, which we set this value to 0.8 (that means that there's an 80% chance that crossover will be applied during the evolution process when generating new offspring.)

Example Run of Our Code:

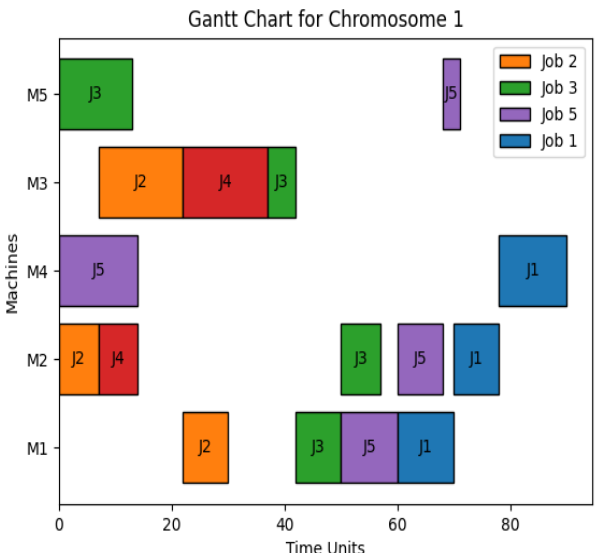
- **Input File:**

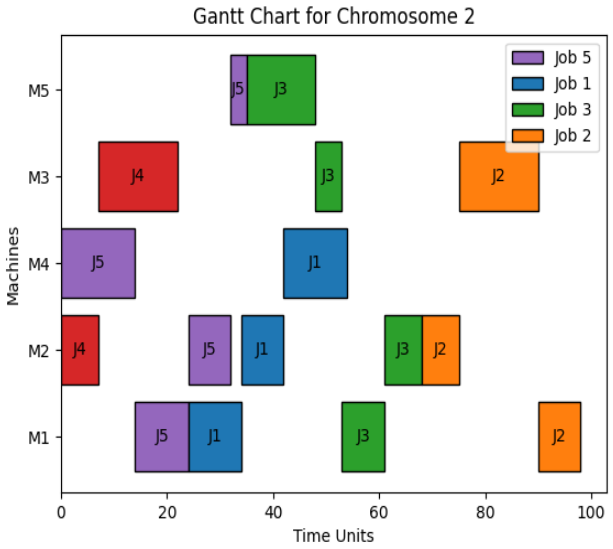
jobShop.py | jobs.csv

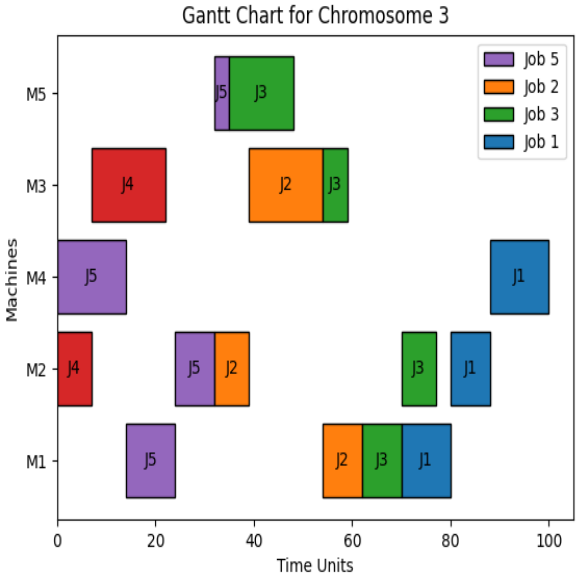
jobs.csv > data

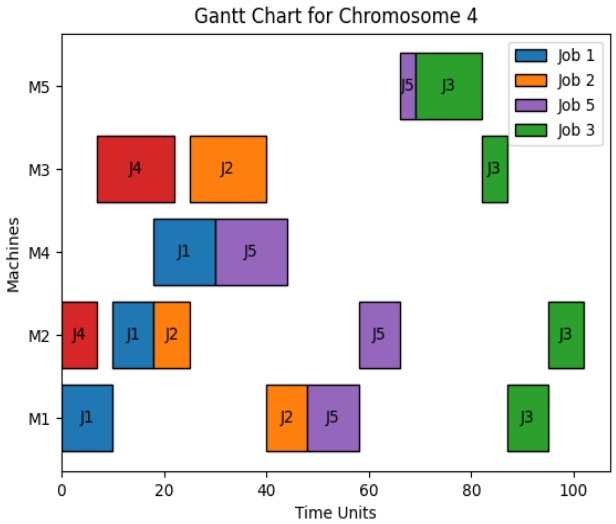
```
1  Job,Operations
2  Job_1,"M1[10]->M2[8]->M4[12]"
3  Job_2,"M2[7]->M3[15]->M1[8]"
4  Job_3,"M5[13]->M3[5]->M1[8]->M2[7]"
5  Job_4,"M2[7]->M3[15]"
6  Job_5,"M4[14]->M1[10]->M2[8]->M5[3]"
7
```

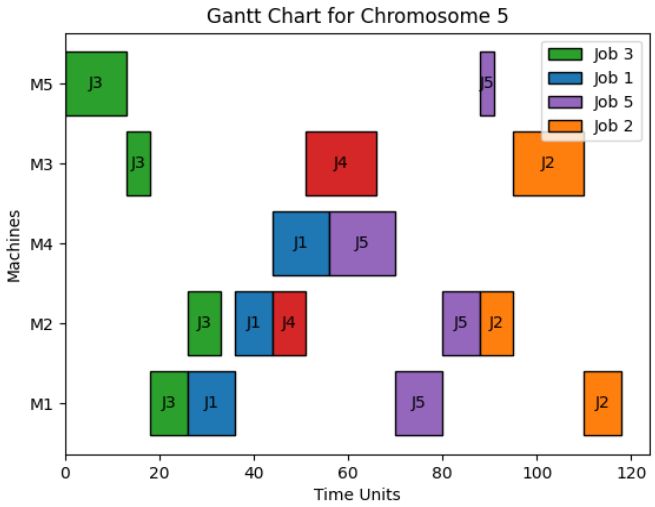
- **Gantt Chart for the solutions for one run:**

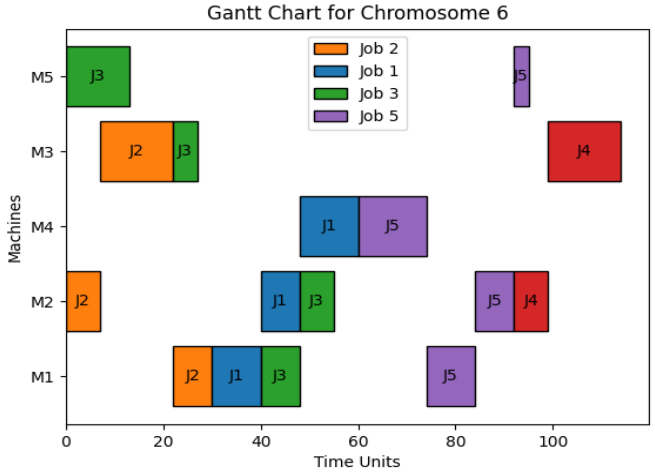












- **Output:**

```
PS C:\Users\Lenovo\Desktop\AI> python -u "c:\Users\Lenovo\Desktop\AI\jobShop.py"
Chromosome 1: [(2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (3, 'M2', 7), (3, 'M3', 15), (0, 'M1', 10), (0, 'M2', 8),
 (0, 'M4', 12), (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3)]
Makespan for Chromosome 1: 85 time units
Chromosome 2: [(2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8), (0, 'M1', 10),
 (0, 'M2', 8), (0, 'M4', 12), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 2: 85 time units
Chromosome 3: [(3, 'M2', 7), (3, 'M3', 15), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (1, 'M2', 7), (1, 'M3', 15)
, (1, 'M1', 8), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12)]
Makespan for Chromosome 3: 85 time units
Chromosome 4: [(3, 'M2', 7), (3, 'M3', 15), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8)
, (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7)]
Makespan for Chromosome 4: 85 time units
Chromosome 5: [(2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (3, 'M2', 7),
 (3, 'M3', 15), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8)]
Makespan for Chromosome 5: 85 time units
Chromosome 6: [(1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (2, 'M5', 13), (2, 'M3', 5)
, (2, 'M1', 8), (2, 'M2', 7), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 6: 85 time units

Best Solution:
[(2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (3, 'M2', 7), (3, 'M3', 15), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12)
, (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3)]
Best Makespan: 85 time units
Fitness value: 1.1764705882352942
```

Note that when we tried another run, we got better solution, such that we have less makespan (less time) and higher fitness:

```
PS C:\Users\Lenovo\Desktop\AI> python -u "c:\Users\Lenovo\Desktop\AI\jobShop.py"
Chromosome 1: [(0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (1, 'M2', 7), (1, 'M3', 15), (1, 'M
1', 8), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 1: 85 time units
Chromosome 2: [(0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (1, 'M2', 7), (1, 'M3', 15), (1, 'M
1', 8), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 2: 85 time units
Chromosome 3: [(0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (1, 'M2', 7), (1, 'M3', 15), (1, 'M
1', 8), (4, 'M4', 14), (4, 'M2', 8), (4, 'M5', 3), (4, 'M1', 10), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 3: 91 time units
Chromosome 4: [(3, 'M2', 7), (3, 'M3', 15), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (1, '
M3', 15), (1, 'M2', 7), (1, 'M1', 8), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7)]
Makespan for Chromosome 4: 70 time units
Chromosome 5: [(3, 'M2', 7), (3, 'M3', 15), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (2, '
M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8)]
Makespan for Chromosome 5: 85 time units
Chromosome 6: [(1, 'M2', 7), (1, 'M3', 15), (1, 'M1', 8), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4
', 12), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (3, 'M2', 7), (3, 'M3', 15)]
Makespan for Chromosome 6: 85 time units

Best Solution:
[(3, 'M2', 7), (3, 'M3', 15), (4, 'M4', 14), (4, 'M1', 10), (4, 'M2', 8), (4, 'M5', 3), (0, 'M1', 10), (0, 'M2', 8), (0, 'M4', 12), (1, 'M3', 15), (1,
'M2', 7), (1, 'M1', 8), (2, 'M5', 13), (2, 'M3', 5), (2, 'M1', 8), (2, 'M2', 7)]
Best Makespan: 70 time units
Fitness value: 1.4285714285714286
PS C:\Users\Lenovo\Desktop\AI>
```