

Using Genetic Algorithm to Optimize Job Shop Scheduling

Case Study: Large-Scale Scheduling Optimization in a Production Line

You are the production manager of a manufacturing plant that produces ten different types of products: A, B, C, D, E, F, G, H, I, and J.

Case 1:

Each product has a specific sequence of machines it must pass through during production. The goal is to optimize the production schedule to minimize the total production time (makespan) and ensure that the machines are utilized efficiently.

Using this algorithm; Jobs 1, 2, ..., 10 represents products A, B, ..., J respectively. Hence Job_id; Job_1 denotes product A, ..., and Job_10 denotes product J. And M1, M2, ..., M5 denotes Machine 1, Machine 2, ... Machine 5 respectively. Then $Mi[j]$ given i ; machine id, and j duration of Mi of a given product Job_id. So Job_1: M1[20] represents producing product A on Machine 1 takes 20 units of duration.

Product	Machine Order
A	1, 2, 3
B	2, 3, 4, 5, 1
C	1, 4, 5, 3, 2
D	2, 5, 1, 4, 3
E	1, 5, 2, 3, 4
F	1, 2, 3, 4
G	1, 4, 2, 5, 3
H	3, 5, 2, 4, 1
I	1, 3, 2, 4, 5
J	3, 2, 4, 1

Product	Machine 1	Machine 2	Machine 3	Machine 4	Machine 5
A	20	30	40	50	60
B	65	25	35	45	55
C	45	55	15	25	35
D	40	50	60	20	30
E	35	45	55	65	25
F	30	60	40	70	50
G	55	35	75	45	65
H	65	45	25	55	35
I	40	60	50	30	70
J	50	30	60	40	20

Input:



Job_1: M1[20] -> M2[30] -> M3[40]
Job_2: M2[25] -> M3[35] -> M4[45] -> M5[55] -> M1[65]
Job_3: M1[45] -> M4[25] -> M5[35] -> M3[15] -> M2[55]
Job_4: M2[50] -> M5[30] -> M1[40] -> M4[20] -> M3[60]
Job_5: M1[35] -> M5[25] -> M2[45] -> M3[55] -> M4[65]
Job_6: M1[30] -> M2[60] -> M3[40] -> M4[70]
Job_7: M1[55] -> M4[45] -> M2[35] -> M5[65] -> M3[75]
Job_8: M3[25] -> M5[35] -> M2[45] -> M4[55] -> M1[65]
Job_9: M1[40] -> M3[50] -> M2[60] -> M4[30] -> M5[70]
Job_10: M3[60] -> M2[30] -> M4[40] -> M1[50]

Population

It is a collection of individual solutions (chromosomes), which are randomly generated initially.

Chromosome Representation

Job_1: M1[20] -> M2[30] -> M3[40], Job_2: M2[25] -> M3[35] -> M4[45] -> M5[55] -> M1[65], ..., Job_10: M3[60] -> M2[30] -> M4[40] -> M1[50]

The chromosome [1, 1, 1, 2, 2, 2, 2, 2, ..., 10, 10, 10, 10] represents the following sequence: Job 1 Operation 1, Job 1 Operation 2, Job 1 Operation 3, Job 2 Operation 1, Job 2 Operation 2, Job 2 Operation 3, Job 2 Operation 4, Job 2 Operation 5,..., Job 10 Operation 1, Job 10 Operation 2, Job 10, Operation 3, Job 10 Operation 4. This structure allows the algorithm to maintain the correct order of operations for each job, while also enabling effective genetic operations like crossover and mutation.

Fitness Function

It determines how good the solution (chromosome) is, and assigns a fitness score for each. Here, it will measure the production time for a machine (lower time means higher fitness score).

- *machine_avail_time*: A dictionary that stores when each machine will be available. It's initialized with all machines available at time 0.
- *job_completion_time*: A dictionary to keep track of when each job is completed. Initially, all jobs have a completion time of 0.
- *job_operation_index*: A dictionary that stores the index of the next operation to be performed for each job. It starts at 0 for all jobs.
- *schedule*: An empty list that will store the detailed schedule of each operation (job, machine, start time, end time).
- It retrieves the details of the job corresponding to the *job_id*.
- It finds the next operation to be performed for that job using *op_index*.
- It determines the machine needed, the *processing_time*, and calculates the *start_time* and *completion_time* for this operation.
- It updates the *machine_avail_time*, *job_completion_time*, and *job_operation_index* accordingly.
- It appends the operation details to the *schedule* list
- *makespan* is the total time taken to complete all jobs (the maximum completion time among all jobs).
- The function returns the *makespan* and the generated *schedule*.

Selection

How to select individuals from population for the next generation (usually select individuals with higher fitness score). In the project, we are picking the best schedules from population based on the fitness function.

Crossover

Combining two parent chromosomes to produce one or more offspring. In our project, the child will take a part from the first parent, and a part from the second parent.

Mutation

It is a genetic operator that introduces small random changes to chromosomes. In the project, we are going to swap chromosomes to create new ones.

Termination Criteria

It terminates when reaching the maximum number of generations.

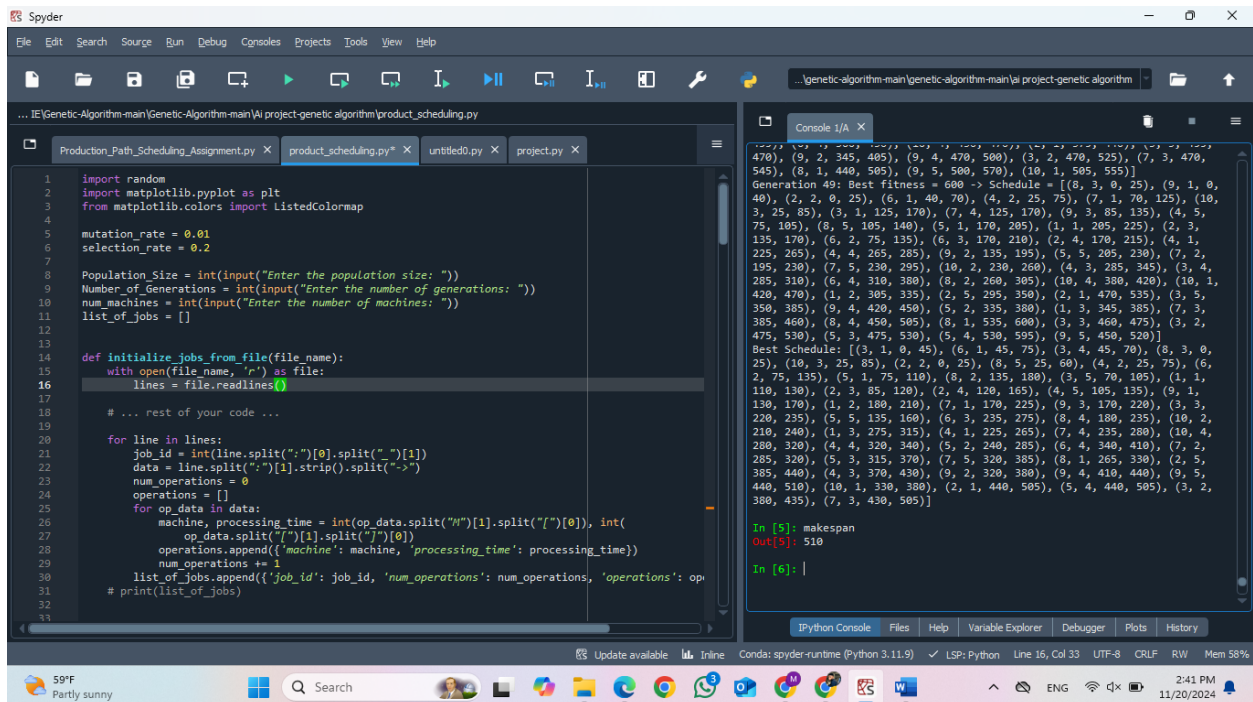
input_file.txt content

```
Job_1: M1[20] -> M2[30] -> M3[40]
Job_2: M2[25] -> M3[35] -> M4[45] -> M5[55] -> M1[65]
Job_3: M1[45] -> M4[25] -> M5[35] -> M3[15] -> M2[55]
Job_4: M2[50] -> M5[30] -> M1[40] -> M4[20] -> M3[60]
Job_5: M1[35] -> M5[25] -> M2[45] -> M3[55] -> M4[65]
Job_6: M1[30] -> M2[60] -> M3[40] -> M4[70]
Job_7: M1[55] -> M4[45] -> M2[35] -> M5[65] -> M3[75]
Job_8: M3[25] -> M5[35] -> M2[45] -> M4[55] -> M1[65]
Job_9: M1[40] -> M3[50] -> M2[60] -> M4[30] -> M5[70]
Job_10: M3[60] -> M2[30] -> M4[40] -> M1[50]
```

Results

For 100 population size, 50 generations and 5 machines

Best fitness 510



```
... IE [Genetic-Algorithm-main] [Genetic-Algorithm-main] \AI project-genetic algorithm\product_scheduling.py
Production_Path_Scheduling_Assignment.py x product_scheduling.py* x untitled0.py x project.py x

1 import random
2 import matplotlib.pyplot as plt
3 from matplotlib.colors import ListedColormap
4
5 mutation_rate = 0.01
6 selection_rate = 0.2
7
8 Population Size = int(input("Enter the population size: "))
9 Number of Generations = int(input("Enter the number of generations: "))
10 num_machines = int(input("Enter the number of machines: "))
11 list_of_jobs = []
12
13 def initialize_jobs_from_file(file_name):
14     with open(file_name, 'r') as file:
15         lines = file.readlines()
16         # ... rest of your code ...
17
18 for line in lines:
19     job_id = int(line.split(":")[0].split("_")[1])
20     data = line.split(":")[1].strip().split("->")
21     num_operations = 0
22     operations = []
23     for op_data in data:
24         machine, processing_time = int(op_data.split(" ")[1].split("(")[0]), int(
25             op_data.split("(")[1].split(")")[0])
26         operations.append({'machine': machine, 'processing_time': processing_time})
27         num_operations += 1
28     list_of_jobs.append({'job_id': job_id, 'num_operations': num_operations, 'operations': op
29 # print(list_of_jobs)
30
31
32
33

Console 1/A x
470), (9, 2, 345, 405), (9, 4, 470, 500), (3, 2, 470, 525), (7, 3, 470,
545), (8, 1, 440, 505), (9, 5, 500, 570), (10, 1, 505, 555)]
Generation 49: Best fitness = 600 -> Schedule = [(8, 3, 0, 25), (9, 1, 0,
40), (2, 2, 0, 25), (6, 1, 40, 70), (4, 2, 25, 75), (7, 1, 70, 125), (10,
3, 25, 85), (3, 1, 125, 170), (7, 4, 125, 170), (9, 3, 85, 135), (4, 5,
75, 105), (8, 5, 105, 140), (5, 1, 170, 205), (1, 1, 205, 225), (2, 3,
135, 170), (6, 2, 75, 135), (6, 3, 170, 210), (2, 4, 170, 215), (4, 1,
225, 265), (4, 4, 265, 285), (9, 2, 135, 195), (5, 5, 205, 230), (7, 2,
195, 230), (7, 5, 230, 295), (10, 2, 135, 195), (4, 3, 285, 345), (3, 4,
285, 310), (6, 4, 310, 380), (8, 2, 260, 305), (10, 4, 380, 420), (10, 1,
420, 470), (1, 2, 305, 335), (2, 5, 295, 350), (2, 1, 470, 535), (3, 5,
350, 385), (9, 4, 420, 450), (5, 2, 335, 380), (1, 3, 345, 385), (7, 3,
385, 460), (8, 4, 450, 505), (8, 1, 535, 600), (3, 3, 460, 475), (3, 2,
475, 530), (5, 3, 475, 530), (5, 4, 530, 595), (9, 5, 450, 520)]
Best Schedule: [(3, 1, 0, 45), (6, 1, 45, 75), (3, 4, 45, 70), (8, 3, 0,
25), (10, 3, 25, 85), (2, 2, 0, 25), (8, 5, 25, 60), (4, 2, 25, 75), (6,
2, 75, 135), (5, 1, 75, 110), (8, 2, 135, 180), (3, 5, 70, 105), (1, 1,
110, 130), (2, 3, 85, 120), (2, 4, 120, 165), (4, 5, 105, 135), (9, 1,
130, 170), (1, 2, 180, 210), (7, 1, 170, 225), (9, 3, 170, 220), (3, 3,
220, 235), (5, 5, 135, 160), (6, 3, 235, 275), (8, 4, 180, 235), (10, 2,
210, 240), (1, 3, 275, 315), (4, 1, 225, 265), (7, 4, 235, 280), (10, 4,
280, 320), (4, 4, 320, 340), (5, 2, 240, 285), (6, 4, 340, 410), (7, 2,
285, 320), (5, 3, 315, 370), (7, 5, 320, 385), (8, 1, 265, 330), (2, 5,
385, 440), (4, 3, 370, 430), (9, 2, 320, 380), (9, 4, 410, 440), (9, 5,
440, 510), (10, 1, 330, 380), (2, 1, 440, 505), (5, 4, 440, 505), (3, 2,
380, 435), (7, 3, 430, 505)]

In [5]: makespan
Out[5]: 510

In [6]:
```



For 500 Population size, 500 generations and 5 machines
Best Fitness/makespan: 505

The screenshot shows the Spyder Python IDE with the following code in the editor:

```

1 import random
2 import matplotlib.pyplot as plt
3 from matplotlib.colors import ListedColormap
4
5 mutation_rate = 0.01
6 selection_rate = 0.2
7
8 Population_Size = int(input("Enter the population size: "))
9 Number_of_Generations = int(input("Enter the number of generations: "))
10 num_machines = int(input("Enter the number of machines: "))
11 list_of_jobs = []
12
13 def initialize_jobs_from_file(file_name):
14     with open(file_name, 'r') as file:
15         lines = file.readlines()
16
17     # ... rest of your code ...
18
19 for line in lines:
20     job_id = int(line.split(":")[0].split("-")[1])
21     data = line.split(":")[1].strip().split("->")
22     num_operations = 0
23     operations = []
24     for op_data in data:
25         machine, processing_time = int(op_data.split("M")[1].split("-")[0]), int(
26             op_data.split("-")[1].split("-")[0])
27         operations.append({'machine': machine, 'processing_time': processing_time})
28         num_operations += 1
29     list_of_jobs.append({'job_id': job_id, 'num_operations': num_operations, 'operations': op
30 # print(list_of_jobs)
31
32
33

```

The console output shows the results of the genetic algorithm:

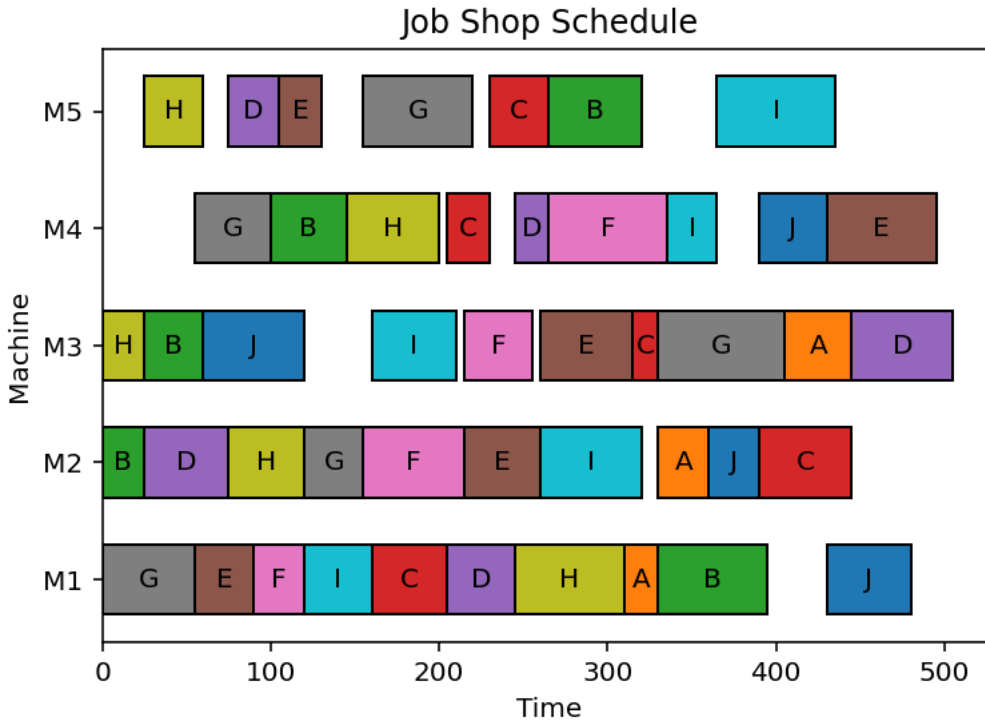
```

555), (3, 2, 450, 505), (10, 1, 415, 465), (7, 3, 450, 525)]
Generation 499: Best fitness = 535 -> Schedule = [(8, 3, 0, 25), (7, 1, 0,
55), (9, 1, 55, 95), (2, 2, 0, 25), (4, 2, 25, 75), (8, 5, 25, 60), (5, 1,
95, 130), (7, 4, 55, 100), (8, 2, 75, 120), (10, 3, 25, 85), (1, 1, 130,
150), (2, 3, 85, 120), (4, 5, 75, 105), (5, 5, 130, 155), (2, 4, 120,
165), (6, 1, 150, 180), (9, 3, 120, 170), (10, 2, 120, 150), (3, 1, 180,
225), (7, 2, 150, 185), (3, 4, 225, 250), (4, 1, 225, 265), (5, 2, 185,
230), (1, 2, 230, 260), (10, 4, 250, 290), (4, 4, 290, 310), (5, 3, 230,
285), (2, 5, 165, 220), (3, 5, 250, 285), (5, 4, 310, 375), (6, 2, 260,
320), (7, 5, 285, 350), (3, 3, 285, 300), (3, 2, 320, 375), (10, 1, 290,
340), (1, 3, 300, 340), (2, 1, 340, 405), (4, 3, 340, 400), (9, 2, 375,
435), (8, 4, 375, 430), (9, 4, 435, 465), (9, 5, 465, 535), (6, 3, 400,
440), (6, 4, 465, 535), (7, 3, 440, 515), (6, 1, 430, 495)]
Best Schedule: [(2, 2, 0, 25), (8, 3, 0, 25), (2, 3, 25, 60), (7, 1, 0,
55), (8, 5, 25, 60), (4, 2, 25, 75), (5, 1, 55, 90), (7, 4, 55, 100), (8,
2, 75, 120), (4, 5, 75, 105), (6, 1, 90, 120), (2, 4, 100, 145), (7, 2,
120, 155), (8, 4, 145, 200), (6, 2, 155, 215), (10, 3, 60, 120), (9, 1,
120, 160), (5, 5, 105, 130), (5, 2, 215, 260), (7, 5, 155, 220), (9, 3,
160, 210), (9, 2, 260, 320), (3, 1, 160, 205), (3, 4, 205, 230), (4, 1,
205, 245), (6, 3, 215, 255), (8, 1, 245, 310), (1, 1, 310, 330), (1, 2,
330, 360), (3, 5, 230, 265), (5, 3, 260, 315), (2, 5, 265, 320), (2, 1,
330, 395), (10, 2, 360, 390), (4, 4, 245, 265), (3, 3, 315, 330), (6, 4,
265, 335), (7, 3, 330, 405), (9, 4, 335, 365), (10, 4, 390, 430), (1, 3,
405, 445), (3, 2, 390, 445), (4, 3, 445, 505), (5, 4, 430, 495), (9, 5,
365, 435), (10, 1, 430, 480)]

In [7]: makespan
Out[7]: 505

In [8]:

```



Conclusion

In this sense genetic algorithm has proven to be effective in optimizing the job shop scheduling by reducing the makespan and after increasing the population size and number of iterations (generations) further reduces the makespan.

References

(*Genetic-Algorithm/Ai project-genetic algorithm/project.py at main · doaahatu/Genetic-Algorithm*, no date; *Google Colab*, no date)

Genetic-Algorithm/Ai project-genetic algorithm/project.py at main · doaahatu/Genetic-Algorithm (no date) *GitHub*. Available at: <https://github.com/doaahatu/Genetic-Algorithm/blob/main/Ai%20project-genetic%20algorithm/project.py> (Accessed: 18 November 2024).

Google Colab (no date). Available at: https://colab.research.google.com/drive/1TQcLiOu_t9Vc_Wi8bJWRwOTU88g30cnh#scrollTo=_K2obFdNeQip (Accessed: 18 November 2024).