HACETTEPE UNIVERSITY
ENGINEERING FACULTY
DEPARTMENT OF ARTIFICIAL INTELLIGENCE ENGINEERING

AIN 300

INTERNSHIP REPORT

Muhammed Beşir ACAR
2220765003

Performed at

Baca Engineering/upu.io

01.07.2024 – 12.08.2024
30 work days

TABLE OF CONTENTS

1 Introduction

I completed my internship at Baca Engineering’s spin-off corporation *upu.io* in the software/IT department. *upu.io* is a next-generation cloud-based IIoT platform that provides end-to-end intelligent manufacturing solutions for production facilities. My main motivation to choose this place is explore and develop the use of AI/ML techniques in production planning. Also, the fact that people I know work here encouraged me to work as a team.

In my internship I developed a multi-objective production scheduling program using genetic algorithms. Objectives include minimizing the makespan, meeting deadlines and maximizing machine utilization. In fact, there are numerous objectives but due to lack of some parameters we did not take any action about the other objectives.

Fast and consistent scheduling while respecting the requested criteria as much as possible is very challenging. This is made autonomous by the algorithms we implemented, saving both time and energy for the company.

2 Company Information

2.1 About the company

Baca Engineering was founded in December 2014. Four years later, in March 2018, the *upu.io* project began development. In December 2020, *upu.io* established an R&D center and completed the first prototype work. *upu.io*'s technological innovations were introduced at *Gitex Global 2022*. A year later, *upu.io* products were launched globally at *Hannover Messe 2023*. Sales started and $300k of sales were made as of October 2023. Series A investment round launched at *CES 2024*. *upu.io* incorporated as a joint-stock company with new investors in March 2024. New *upu.tower* and *upu.kwh* products will be launched at *Hannover Messe 2024*.

upu.io is currently located in Teknopark Ankara.

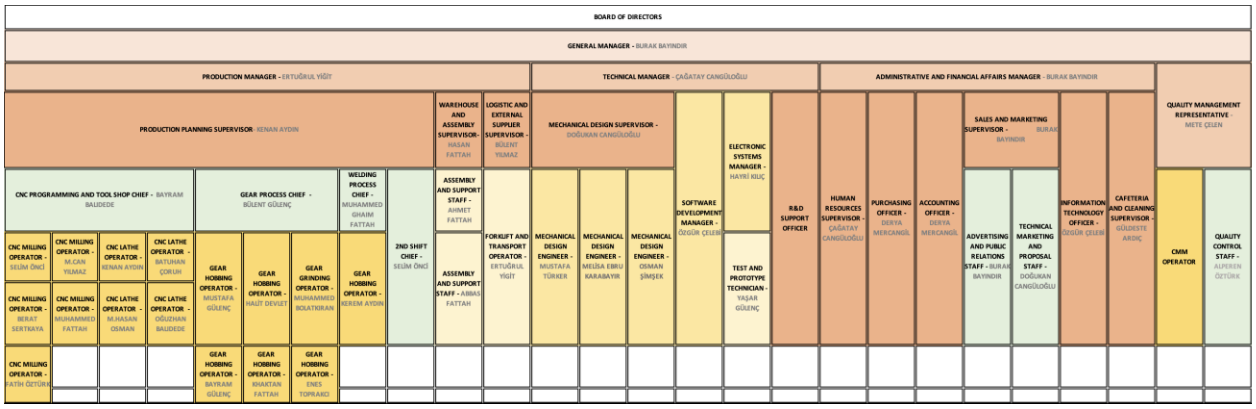

BOARD OF DIRECTORS
GENERAL MANAGER - BURAK BAYINDIR
PRODUCTION MANAGER - ERTUĞRUL YİĞİT
TECHNICAL MANAGER - ÇAĞATAY CANGÜLOĞLU
ADMINISTRATIVE AND FINANCIAL AFFAIRS MANAGER - BURAK BAYINDIR
QUALITY MANAGEMENT REPRESENTATIVE - METE ÇELEN
PRODUCTION PLANNING SUPERVISOR - KENAN AYDIN
WAREHOUSE AND ASSEMBLY SUPERVISOR- HASAN FATTAH
LOGISTIC AND EXTERNAL SUPPLIER SUPERVISOR - BÜLENT YILMAZ
MECHANICAL DESIGN SUPERVISOR - DOĞUKAN CANGÜLOĞLU
SOFTWARE DEVELOPMENT MANAGER - ÖZGÜR ÇELEBİ
ELECTRONIC SYSTEMS MANAGER - HAYRİ KILIÇ
R&D SUPPORT OFFICER
HUMAN RESOURCES SUPERVISOR - ÇAĞATAY CANGÜLOĞLU
PURCHASING OFFICER - DERYA MERCANGİL
ACCOUNTING OFFICER - DERYA MERCANGİL
SALES AND MARKETING SUPERVISOR - BURAK BAYINDIR
INFORMATION TECHNOLOGY OFFICER - ÖZGÜR ÇELEBİ
CAFETERIA AND CLEANING SUPERVISOR - GÜLDESTE ARDIÇ
CNC PROGRAMMING AND TOOL SHOP CHIEF - BAYRAM BALIDEDE
GEAR PROCESS CHIEF - BÜLENT GÜLENÇ
WELDING PROCESS CHIEF - MUHAMMED GHAIM FATTAH
2ND SHIFT CHIEF - SELİM ÖNCİ
ASSEMBLY AND SUPPORT STAFF - AHMET FATTAH
ASSEMBLY AND SUPPORT STAFF - ABBAS FATTAH
FORKLIFT AND TRANSPORT OPERATOR - ERTUĞRUL YİĞİT
MECHANICAL DESIGN ENGINEER - MUSTAFA TÜRKER
MECHANICAL DESIGN ENGINEER - MELİSA EBRU KARABAYIR
MECHANICAL DESIGN ENGINEER - OSMAN ŞİMŞEK
TEST AND PROTOTYPE TECHNICIAN - YAŞAR GÜLENÇ
ADVERTISING AND PUBLIC RELATIONS STAFF - BURAK BAYINDIR
TECHNICAL MARKETING AND PROPOSAL STAFF - DOĞUKAN CANGÜLOĞLU
CMM OPERATOR
QUALITY CONTROL STAFF - ALPEREN ÖZTÜRK
CNC MILLING OPERATOR - SELİM ÖNCİ
CNC MILLING OPERATOR - M.CAN YILMAZ
CNC LATHE OPERATOR - KENAN AYDIN
CNC LATHE OPERATOR - BATUHAN ÇORUH
CNC MILLING OPERATOR - BERAT SERTKAYA
CNC MILLING OPERATOR - MUHAMMED FATTAH
CNC LATHE OPERATOR - M.HASAN OSMAN
CNC LATHE OPERATOR - OĞUZHAN BALIDEDE
CNC MILLING OPERATOR - FATİH ÖZTÜRK
GEAR HOBBING OPERATOR - MUSTAFA GÜLENÇ
GEAR HOBBING OPERATOR - HALİT DEVLET
GEAR GRINDING OPERATOR - MUHAMMED BOLATKIRAN
GEAR HOBBING OPERATOR - KEREM AYDIN
GEAR HOBBING OPERATOR - BAYRAM GÜLENÇ
GEAR HOBBING OPERATOR - KHAKTAN FATTAH
GEAR HOBBING OPERATOR - ENES TOPRAKCI

Fig. 1. Organizational chart

Baca Engineering has a wide range of products that can meet all the needs of the defense industry, oil and mineral drilling, machinery manufacturing, gear manufacturing and reducer manufacturing sectors. Baca Engineering carries out contract manufacturing services, serial and prototype metal part manufacturing in accordance with AS9100 aviation standards, in line with customer requests.

Due to the company's privacy policy, stakeholder informations are not shared.

2.2 About the department

In the department/spin-off company *upu.io* where I completed my internship there was no available organizational chart but, there were two backend developers, four frontend developers (six computer engineers in total), two UI/UX designers, one industrial engineer, four electrical and electronics engineers, and two assembly engineers.

I did not have a specific position. I can say that I completed my study in contact with staff from all branches, especially industrial engineers and developers.

There are two different categories of products in the department: *upu.sense* and *upu.tower*.

upu.sense are IIOT sensor sets compatible with all machines.

upu.current	0-500 A interleaved sensitive intelligent current sensor with 4-20 mA analog outputs.
upu.prox	PNP and NPN intelligent IIOT proximity sensor.
upu.ldr	Intelligent IIOT light sensor with adjustable analog and digital outputs.
upu.vibra	AI supported IIOT vibration sensor with 4-20 mA output for predictive maintenance activities.
upu.temp	Intelligent and sensitive IIOT temperature sensor with 4-20 mA analog output.
upu.press	Intelligent and sensitive IIOT pressure sensor with 4-20 mA analog output.

upu.tower processes all machine data with AI-supported sensor fusion and easily connects to the system with its own sensor sets without intervening in the machines and transmits this data to employees without the need for a gateway.

2.3 About the hardware and software systems

Existing hardware and software systems were not shared due to the company's privacy policy.

2.4 About the supervisor

My supervisor was Hayri KILIÇ, his address is İlkyerleşim Mah., 1237. Cad, No:49 Yenimahalle/ANKARA. His phone number is +90 (534) 014 43 54. His email address is hk@hayrikilic.net. Information about the education is below:

- Bachelor’s degree: Kırıkkale University, Faculty of Engineering, Electrical and Electronics Engineering, 2020
- Bachelor’s degree: Anadolu University, Business School, Business, 2021
- MS degree: Konya Technical University, Graduate Education Institute, Industrial Engineering, 2023
- MS degree: Selçuk University, Institute of Science, Electrical and Electronics Engineering, 2023
- MS degree: Konya Technical University, Graduate Education Institute, Computer Engineering, 2024

You can reach his studies from this link: www.hayrikilic.net.

3 Work Done

Within the scope of the product we developed in the project (Tübitak 1501), analyses are carried out in line with the data obtained from machine, personnel and product-oriented informations. In order to manage production planning in a holistic and integrated manner and to create production plans automatically, modeling should be performed using various AI/ML algorithms. In this context, there is a need to automatically perform activities such as capacity management, scheduling, resource allocation, and to facilitate the decision-making process by providing suggestions to the user as a result of these studies.

The developments in the project to date have been carried out by me. After the studies mentioned in this report, improvements such as predictive maintenance applications, inventory optimization, integration with IIoT platforms etc. are scheduled to be done. The proposed project has the potential to give the company a global reputation and move upmarket.

3.1 Methodology

Suppose there is a set of n jobs $\{J_1, J_2, J_3, \dots, J_n\}$ and a set of machines $M = \{M_1, M_2, M_3, \dots, M_m\}$. Each job consists of a set of operations $\{O_{i1}, O_{i2}, O_{i3}, \dots, O_{in_i}\}$, where is the number of operations that occur. Each operation O_{ij} $(i = 1, 2, \dots, n; j = 1, 2, \dots, n_i)$ must be processed by a machine from the given set of machines. The problem is therefore to determine both an assignment and a sequence of operations on machines to satisfy some criteria [1]. So, the problem consists of two sub-problems: machine selection (MS) problem and operation sequencing (OS) problem. Determined constraints are as follows:

(1) Jobs are independent. Preemption of jobs is not allowed, and each machine can only perform one operation at a time.
(2) Different operations of a job cannot be processed simultaneously.
(3) All jobs and machines are available at time zero.
(4) After a job is processed on a machine, it is immediately transferred to the next machine, assuming the transfer time is negligible.
(5) The setup time for operations on machines is independent of the processing order and is included in the processing time.

The genetic algorithm begins by setting its parameters and initializing the population, with the generation count (*Gen*) starting at 1. Each individual in the population is then evaluated concurrently based on the objective function. The algorithm checks whether the termination criteria have been met; if so, it proceeds to output the best solution. If not, a new population is generated using genetic operators such as selection, crossover, and mutation. The generation count is incremented by 1, and the evaluation process repeats until the termination criteria are satisfied, after which the best solution is provided [5].

3.1.1 Encoding and decoding

In the context of the FJSP, a chromosome represents a potential solution, and its encoding is essential for the GA. In this case, the chromosome consists of two parts: the OS (Operation Sequence) string and the MS (Machine Sequence) string [2], each with distinct encoding methods.

The OS string uses an *operation-based representation* method, which consists of job numbers. If we have n jobs, the OS string is an unpartitioned permutation of the job numbers where each job i appears On_i times. On_i represents the total number of operations for job i.

When scanning the OS string from left to right, the *f*-th appearance of job number *i* corresponds to the *f*-th operation of that job. For instance, the first occurrence of job *i* corresponds to its first operation, the second occurrence corresponds to its second operation, and so on. The crucial property of this encoding is that any random permutation of the OS string can be decoded into a feasible solution.

The length of the OS string is equal to $\sum_{i=1}^{n} On_i$, the sum of all operations across all jobs. The initial OS population is generated randomly but respects this encoding rule.

The MS string represents the specific machines chosen for each operation of the jobs. Its length is also equal to $\sum_{i=1}^{n} On_i$, just like the OS string. The MS string is divided into n parts, where each part corresponds to a job. The length of the i-th part of the MS string is On_i, which is the number of operations for job i.

Each part of the MS string indicates the machines selected for each operation of that particular job. Each gene in this string corresponds to the machine chosen for a specific operation, and this assignment remains fixed throughout the search process.

For example, suppose the h-th operation of job i can be processed by a set of machines $S_{ih} = \{m_{ih1}, m_{ih2}, \dots, m_{ihc_{ih}}\}$, where c_{ih} is the number of available machines for that operation. The i-th part of the MS string can be represented as $\{g_{i1}, g_{i2}, \dots, g_{ih}, \dots, g_{iOn_i}\}$, where each g_{ih} is an integer between 1 and c_{ih}. This means that the h-th operation of job i is assigned to the g_{ih}-th machine $m_{ihg_{ih}}$ from the set S_{ih} [1].

The initial MS string is generated by randomly selecting a machine from the available machine set for each operation of every job.

The decoding process focuses on assigning machines to operations and determining their start and completion times based on both regular and irregular criteria such as makespan and deadlines [3, 4]. The notations used to explain the decoding procedure are described below :

n	the total number of jobs
m	the total number of machines;
o_{ij}	the jth operation of the ith job;
as_{ij}	the allowable starting time of operation o_{ij};
s_{ij}	the earliest starting time of operation o_{ij};
k	the alternative machine corresponding to o_{ij};
t_{ijk}	the processing time of operation o_{ij} on machine k;
c_{ij}	the earliest completion time of operation o_{ij}, i.e. $c_{ij} = s_{ij} + t_{ijk}$;

Decoding Steps:

1. Use the MS string to allocate a specific machine for each operation.

2. For each machine, determine the set of operations it will process. Let the set for machine 'a' be:
$$m_a = \{o_{ij}\} 1 \leq a \leq m$$
3. For each job, identify the machines on which its operations will be performed. Let the set for job 'd' be:
$$Jm_d = \{machine\} 1 \leq d \leq n$$
4. The allowable start time as_{ij} of operation o_{ij} depends on the completion time of its preceding operation $o_{i(j-1)}$ in the same job. For operations in set m_a, this becomes:
$$as_{ij} = c_{i(j-1)}$$
5. For the machine assigned to operation o_{ij}, examine its available idle periods. These periods are defined as $[t_s, t_e]$. The earliest start time s_{ij} for o_{ij} is determined as follows [3]:
```
if max($as_{ij}$, $t_s$) + $t_{ijk}$ ≤ $t_e$:
    $s_{ij} = t_s$
else:
    find the next available area (time window)
    if no area satisfies the condition:
        $s_{ij}$ = max$(as_{ij}, c(o_{ij} - 1))$
($c(o_{ij} - 1)$ is the completion time of the pre-operation of $o_{ij}$ for the same machine);
```
6. Once the start time is determined, calculate the completion time for operation o_{ij} using the formula: $c_{ij} = s_{ij} + t_{ijk}$;
7. For each job, generate a set of start times and completion times for its operations:
$$T_d(s_{ij}, c_{ij}) 1 \leq d \leq n$$

3.1.2 Genetic Operators

In GA, effective genetic operators are crucial for addressing the problem at hand and efficiently producing high-quality individuals within the population. These operators can generally be categorized into three main types: selection, crossover, and mutation.

Selection

The selection operator in GA is responsible for choosing individuals based on their fitness levels. In this study, two selection strategies are employed:

a) **Elitist Selection**: This method ensures that a certain number of the fittest individuals, calculated as $p_r \times Popsize$ (where p_r is the reproduction probability and $Popsize$ is the population size), are carried over from the parents to the next generation.

b) **Tournament Selection**: Here, a group of individuals is randomly chosen from the population, with the group size determined by a parameter b, which in this study is set to 2. The individual with the highest fitness among the group is selected. This selection allows for a balance between exploration and exploitation by adjusting the tournament size.

The fitness value F consists of three different criterions:
(1) Makespan, which is the maximum time taken by any machine to complete all its assigned jobs. Fitness calculation in makespan is done by decoding the operations and machines schedules and finding the latest job completion time on each machine.
(2) Tardiness, which is the total delay of jobs that finish after their deadlines. Calculation consists of decoding the schedules and comparing each job's completion time to its deadline.
(3) Utilization of the machines, that can differentiate according to the sum of the idle times of the machines.
Which metrics the requested schedule focuses on is determined by the weight values assigned to the criterions. For the set of criterions $\{q_1, q_2, q_3, \dots, q_n\}$ and set of weights $\{w_1, w_2, w_3, \dots, w_n\}$ the following equation is determined:

$$F = \sum_{i=1}^{n} w_i q_i$$

Here, the weights w_i must satisfy the condition: $\sum_{i=1}^{n} w_i = 1$. This equation ensures that the weights are normalized and the fitness value F is a weighted sum of the criterions.

When the fitness value F between the best individuals of any two generations remains constant for 20 (specified in Fig. 2 as *maxStagnantStep*) generations [1], the GA is terminated assuming no further improvement can be demonstrated.

Crossover
In this study, two different crossover operators were implemented for the OS string in the GA. During the GA process, one of these crossover operators is selected at random with an equal probability (50%) to perform the crossover. The first operator used is the Precedence Operation Crossover (POX) [6], which is designed to maintain the order of precedence between operations across parent and offspring solutions. The second operator is the Job-Based Crossover (JBX) [7], which takes a different approach by grouping jobs from the parents into distinct sets and swapping them between offspring.

For the MS string, a two-point crossover method has been employed as the chosen crossover operator. In this approach, two random positions are selected within the parent strings. The offspring are then produced by swapping the elements between these two positions in both parent strings. Since each gene in the MS string represents a specific machine assigned to a fixed operation, the operation sequence remains consistent throughout the search process. This crossover technique ensures that the offspring are valid solutions if the selected parents are feasible.

Mutation

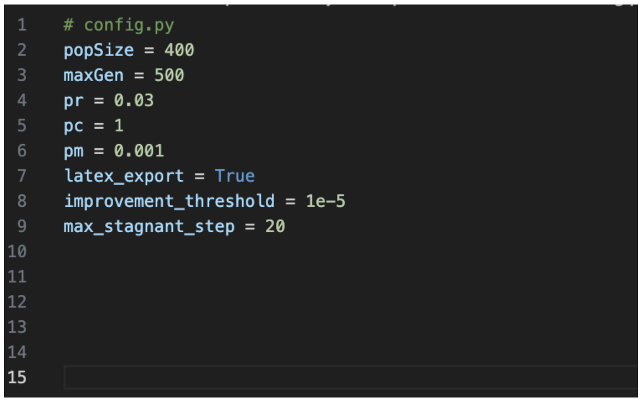
In this study, two different mutation operators have been applied to the OS string. During the GA process, one of these mutation operators is randomly selected with equal probability (50%) to perform the mutation. The first operator is the swapping mutation, which involves exchanging the positions of selected elements within the string. The second operator is the neighborhood mutation, which explores neighboring configurations by altering specific elements in the string.

For the MS string, a dedicated mutation operator has been implemented. This operator involves selecting multiple positions within the parent string and modifying the machine assignments for the corresponding operations.

3.2 Implementation

The proposed GA procedure was coded in Python and implemented in a computer with four high-performance 3.49 GHz and four energy-efficient 2.42 GHz cores with 16.0 GB of RAM memory.

In order to visualize the activity of proposed GA, the last eight order data of the company in which the study was conducted were taken into consideration. As mentioned before this study consists of more than one criterion: makespan, tardiness and machine utilization.



```
# config.py
popSize = 400
maxGen = 500
pr = 0.03
pc = 1
pm = 0.001
latex_export = True
improvement_threshold = 1e-5
max_stagnant_step = 20
```

Fig. 2. config.py where the parameters are inititalized.

In the proposed algorithm, the GA terminates when the number of generations reaches to the maximum value (specified as *maxGen* in Fig. 2) or the permitted maximum step size with no improving (specified as *max_stagnant_step* in Fig. 2).

3.2.1 main.py

The main function in our script orchestrates the entire GA process. The function initially reads the command-line arguments which are the order information dataset, input JSON file, output JSON file, method index and weights of the criterions. Then the population is initialized by calling the '`encoding.initializePopulation`' and operators selection, crossover, and mutation are called to evolve the population over generations until a termination condition is met. Finally the best solution is decoded in the '`decoding`' class, missed deadlines are checked and results are exported as needed.

In order to replicate the preferences, we set method index from 0 to 4. For instance if method index is 0, the algorithm only focuses on minimizing the makespan and if it is 1, the only perspective is tardiness.

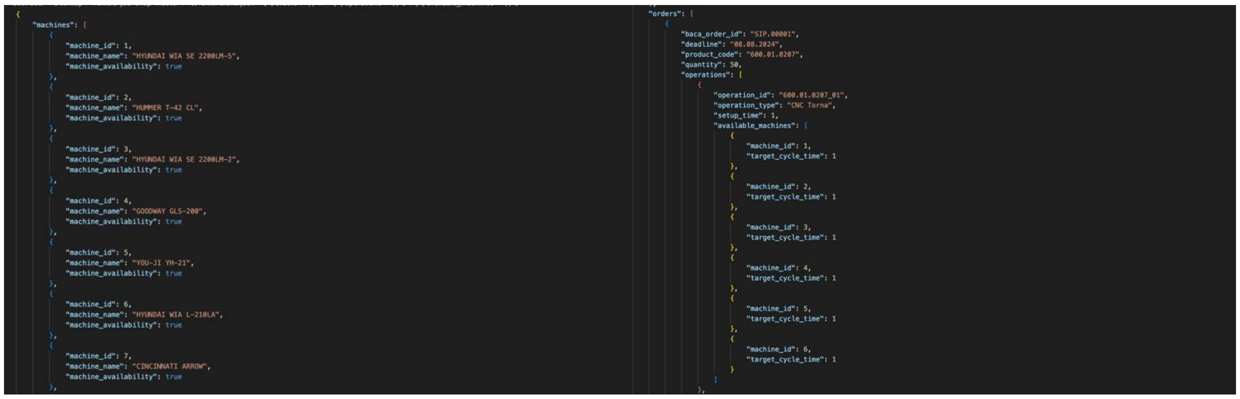
Algorithm 1 While loop that continues until the termination condition is met

```
1.  pool ← multiprocessing pool (mp.Pool) with CPU count
2.  fitness_values ← [ ]
3.  no_improvement_count ← 0
4.  while gen ≤ maxGen and not shouldTerminate(population, gen) do
5.      population ← selection(population, parameters, methodIndex, weights)
6.      population ← crossover(population, parameters)
7.      population ← mutation(population, parameters)
8.
9.      fitness_results ← poolStarmap(evaluate_fitness, [(ind, parameters, methodIndex, weights) for ind in
population])
10.
11.     current_best_fitness ← min(fitness_results)
12.     fitness_values ← fitness_values ∪ {current_best_fitness}
13.     if len(fitness_values) > 1 then
14.         previous_best_fitness ← fitness_values[-2]
15.
16.     if |previous_best_fitness - current_best_fitness| < improvement_threshold then
17.         no_improvement_count += 1
18.     else
19.         no_improvement_count ← 0
20.
21.     if no_improvement_count ≥ max_stagnant_step then
22.         break
23.
24. print "Generation", gen, "completed"
25. gen += 1
```

Fitness evaluation can be computationally expensive when the population size is large. To avoid a little bit of that, multiprocessing pool '`mp.Pool`' is used (as can be seen in the Algorithm 1) to distribute the evaluation of fitness across all available CPU cores. By distributing these evaluations, %50 speedup observed in the GA for the method index 4 (weighted) when the population size is less than 500.

3.2.2 excel_reader.py

This script receives the Excel file containing various sheets related to orders, products, operations and stations (machines) of the company. It processes this data to determine available machines for each operation and merges order details based on specific criteria.



```json
{
    "machines": [
        {
            "machine_id": 1,
            "machine_name": "HYUNDAI WIA SE 2200LM-5",
            "machine_availability": true
        },
        {
            "machine_id": 2,
            "machine_name": "HUMMER T-42 CL",
            "machine_availability": true
        },
        {
            "machine_id": 3,
            "machine_name": "HYUNDAI WIA SE 2200LM-2",
            "machine_availability": true
        },
        {
            "machine_id": 4,
            "machine_name": "GOODWAY GLS-200",
            "machine_availability": true
        },
        {
            "machine_id": 5,
            "machine_name": "YOU-JI YH-21",
            "machine_availability": true
        },
        {
            "machine_id": 6,
            "machine_name": "HYUNDAI WIA L-210LA",
            "machine_availability": true
        },
        {
            "machine_id": 7,
            "machine_name": "CINCINNATI ARROW",
            "machine_availability": true
        },
```

```json
    "orders": [
        {
            "baca_order_id": "SIP.00001",
            "deadline": "08.08.2024",
            "product_code": "600.01.0207",
            "quantity": 50,
            "operations": [
                {
                    "operation_id": "600.01.0207_01",
                    "operation_type": "CNC Torna",
                    "setup_time": 1,
                    "available_machines": [
                        {
                            "machine_id": 1,
                            "target_cycle_time": 1
                        },
                        {
                            "machine_id": 2,
                            "target_cycle_time": 1
                        },
                        {
                            "machine_id": 3,
                            "target_cycle_time": 1
                        },
                        {
                            "machine_id": 4,
                            "target_cycle_time": 1
                        },
                        {
                            "machine_id": 5,
                            "target_cycle_time": 1
                        },
                        {
                            "machine_id": 6,
                            "target_cycle_time": 1
                        }
                    ]
                },
```

Fig. 3. Image of the input JSON file.

The processed data is then exported to a JSON file (shown in Fig. 3), making it easier to use for further processing, analysis, or integration with other systems.

3.2.3 parser.py

This code defines a function '`parse(path)`' that reads JSON file, visualized in Fig. 3, from the specified path, processes the data contained within it and returns various extracted details. First JSON data is loaded and lists are initialized to store information about jobs, deadlines, machine details, company order IDs and product codes. For each job in the orders, the deadline in minutes from the current time is calculated, the *quantity*, *company_order_id* and *product_code* are extracted and each operation is processed. For each operation, the appropriate machines are identified, processing times are calculated according to the operation type and quantity, and this data is compiled into a job list. Finally, a dictionary is returned containing the total number of machines, the list of jobs, deadlines, company order IDs, product codes, machine availability and machine names.

For simplicity, the processing time of any operation is given equally for all the machines available for that operation. In addition, a new machine entry is created for every '*Dış Proses*' which are the external operations.

3.2.4 encoding.py

This module has three main functions: `generateOS`, `generateMS`, and `initializePopulation`. With `generateOS`, a list (OS) is generated representing the order in which operations are performed based on the job parameters provided from `parser.py`. By iterating over jobs and related operations, the job indexes are added to the OS list and then shuffled randomly.

Algorithm 2 Encoding process

```
generateOS(parameters):
    jobs ← parameters['jobs']
    OS ← [ ]
    i ← 0

    for each job in jobs
        for each operation in job
            OS ← OS ∪ {i}
        i ← i + 1

    Randomly shuffle OS

    return OS

generateMS(parameters):
    jobs ← parameters['jobs']
    MS ← [ ]

    for each job in jobs
        for each operation in job
            randomMachine ← randomInteger(0, len(operation) - 1)
            MS ← MS ∪ {randomMachine}

    return MS

initializePopulation(parameters):
    gen1 ← [ ]

    for i from 0 to (popSize – 1)
        OS ← generateOS(parameters)
        MS ← generateMS(parameters)
        gen1 ← gen1 ∪ {(OS, MS)}

    return gen1
```

`generateMS`, on the other hand, generates a list (MS) of randomly selected machine indices for each process based on the machine availability specified in the dictionary returned from `parser.py`. Finally, with `initializePopulation`, an initial population of chromosomes `‘gen1’` is generated, each represented as a tuple containing a list of OS and MS as shown in Algorithm 2.

3.2.5 genetic.py

This code defines several fitness functions to evaluate solutions based on four criterions mentioned before: makespan, deadline and machine utilization. It includes selection mechanisms (elitist and tournament selection) to choose individuals for the next generation, crossover operators (precedence operation a.k.a. POX, job-based a.k.a. JBX, and two-point crossover) to combine solutions, and mutation operators (swapping, neighborhood, and half mutation) to introduce variability. The overall goal is to evolve the population over multiple generations.

The selection function creates a new population by first selecting the fittest individuals (`elitistSelection`) and then filling up the remaining slots with individuals selected through `tournamentSelection`. This ensures that the best solutions are preserved while introducing diversity.

Algorithm 3 Selection operator

```
1. selection(population, parameters, methodIndex, weights):
2.    newPop ← elitist_selection(population, parameters, methodIndex, weights)
3.
4.    while length(newPop) < length(population) do
5.      newPop ← newPop ∪ {tournament_selection(population, parameters, methodIndex, weights)}
6.
7.    return newPop
```

Algorithm 4 Crossover operator

```
1. crossover_OS(p1, p2, parameters)
2.    if random_choice([True, False]) then
3.        return precedence_operation_crossover(p1, p2, parameters)
4.    else
5.        return job_based_crossover(p1, p2, parameters)
6.
7. crossover_MS(p1, p2)
8.    return two_point_crossover(p1, p2)
9.
10. crossover(population, parameters)
11.    newPop ← ( )
12.    i ← 0
13.
14.    while i < length(population) do
15.        (OS1, MS1) ← population[i]
16.        (OS2, MS2) ← population[i+1]
17.
18.        if random() < config.pc then
19.            (oOS1, oOS2) ← crossover_os(OS1, OS2, parameters)
20.            (oMS1, oMS2) ← crossover_ms(MS1, MS2)
21.            newPop ← newPop ∪ {(oOS1, oMS1)}
22.            newPop ← newPop ∪ {(oOS2, oMS2)}
```

```
23.     else
24.         $newPop \leftarrow newPop \cup \{(oOS1, oMS1)\}$
25.         $newPop \leftarrow newPop \cup \{(oOS2, oMS2)\}$
26.
27.     $i \leftarrow i + 2$
28.
29.   return newPop
```

Algorithm 5 Mutation operator

```
1. mutation_OS(p)
2.   if random_choice([True, False]) then
3.       return swapping_mutation(p)
4.   else
5.       return neighborhood_mutation(p)
6.
7. mutation_MS(p, parameters)
8.   return half_mutation(p, parameters)
9.
10. mutation(population, parameters)
11.   $newPop \leftarrow ()$
12.
13.   for each (OS, MS) in population do
14.     if random() < config.pm then
15.         $oOS \leftarrow$ mutation_OS(OS)
16.         $oMS \leftarrow$ mutation_MS(MS, parameters)
17.         $newPop \leftarrow newPop \cup \{(oOS, oMS)\}$
18.     else
19.         $newPop \leftarrow newPop \cup \{(OS, MS)\}$
20.
21.   return newPop
```

The crossover function generates new solutions by combining parts of two parent solutions. It uses two types of crossover operations: `crossoverOS` (which chooses randomly between POX and JBX) and `crossoverMS` (which uses a two-point crossover). These operations are applied with a probability defined in the Fig. 2 (`config.py`) as `‘pc’`.

The mutation function introduces diversity. It gives random movement about the search space, thus preventing the GA becoming trapped in ‘blind corners’ or ‘local optima’ during the search. `mutationOS` decides randomly between swapping or neighborhood mutations for the operation sequence, while `mutationMS` applies a half-mutation for the machine sequence. Mutation occurs with a probability defined in the Fig. 2 (`config.py`) as `‘pm’`. The implementation logic of the operators is shown in the Algorithm 3, 4 and 5 above.

3.2.6 decoding.py

The decoding class is responsible for translating the encoded solutions (i.e., OS and MS) into meaningful schedules and performance metrics.

The operation sequence (OS) and machine sequence (MS) parameters are taken and converted into machine operations and job completion times. The order in which operations are processed on each machine, the start and finish times of each operation and the utilized machine are determined. In addition, calculated job completion times are compared with the deadlines specified in the parameters to check whether any job has exceeded the deadline. Then the orders that missed deadlines and how much they missed are reported in the console. Also, the frequency of use of the machines is computed by calculating the idle time of each machine from the current date to the last operation performed by that machine. Finally decoded machine operations are converted into a format suitable for graphical representation (in this study Gantt chart) and data is prepared for visualization of the program.

3.3 Visualization and Analysis

3.3.1 gantt.py

The gantt module is primarily responsible for visualizing and exporting scheduling information which is translated in `decoding.py`, using Gantt chart in *Plotly* library.

Here is the sample outputs for the specified parameters:

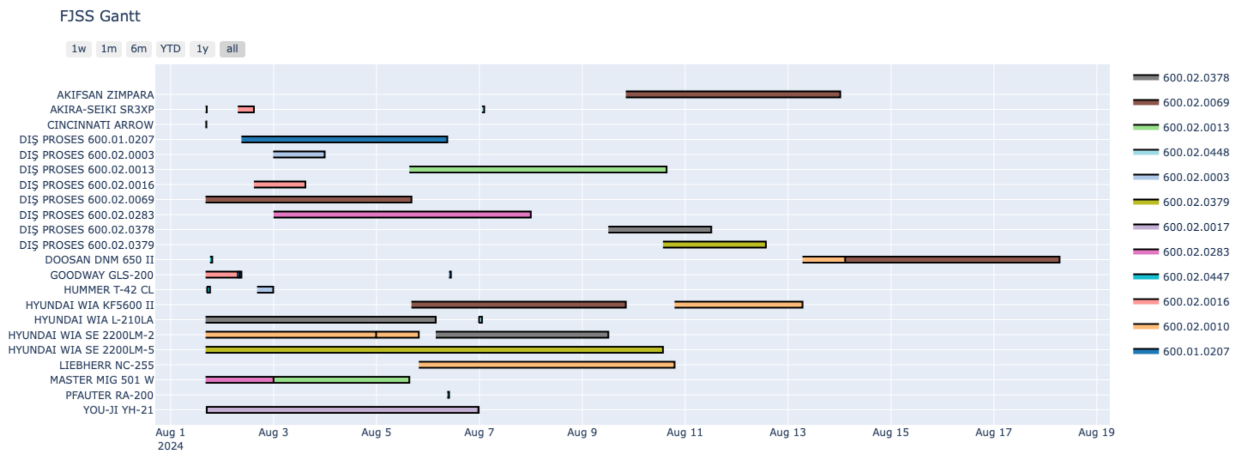
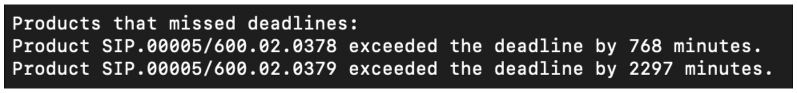


Fig. 4. Received output for *methodIndex* = 1 (tardiness criterion), *popSize* = 400, *maxGen* = 500, *pr* = 0.03, *pc* = 1, *pm = 0.001*, *improvement_threshold* = 0.00001 and *max_stagnant_step* = 20.



```
Products that missed deadlines:
Product SIP.00005/600.02.0378 exceeded the deadline by 768 minutes.
Product SIP.00005/600.02.0379 exceeded the deadline by 2297 minutes.
```

Fig. 5. Console output for the orders that missed deadlines.

The algorithm was executed on August 1, 2024 at 4:15 PM. The deadlines for the products given on the right side in the Fig. 4 are 11.08.2024, 21.08.2024, 11.08.2024, 31.08.2024, 08.08.2024, 11.08.2024, 08.08.2024, 11.08.2024, 31.08.2024, 08.08.2024, 22.08.2024 and 09.08.2024 respectively. The exceedings are given in Fig. 5.

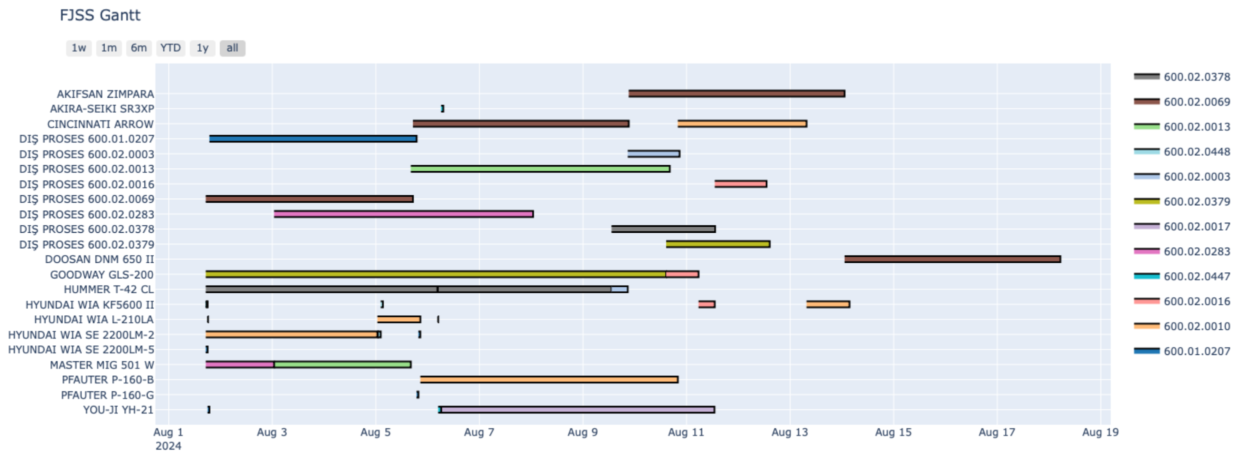


Fig. 6. Received output for *methodIndex* = 0 (makespan criterion), *popSize* = 400, *maxGen* = 500, *pr* = 0.03, *pc* = 1, *pm = 0.001*, *improvement_threshold* = 0.00001 and *max_stagnant_step* = 20.

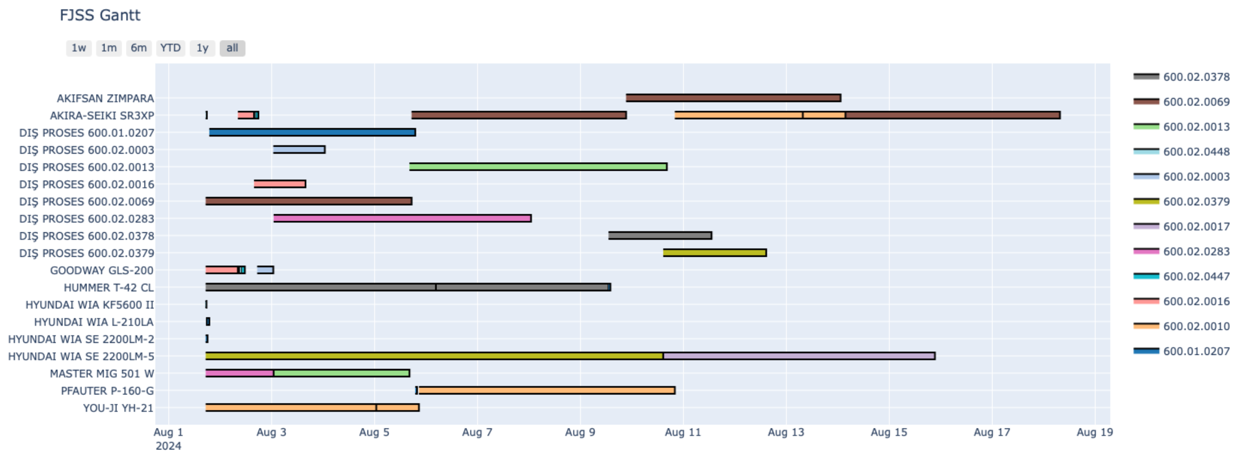
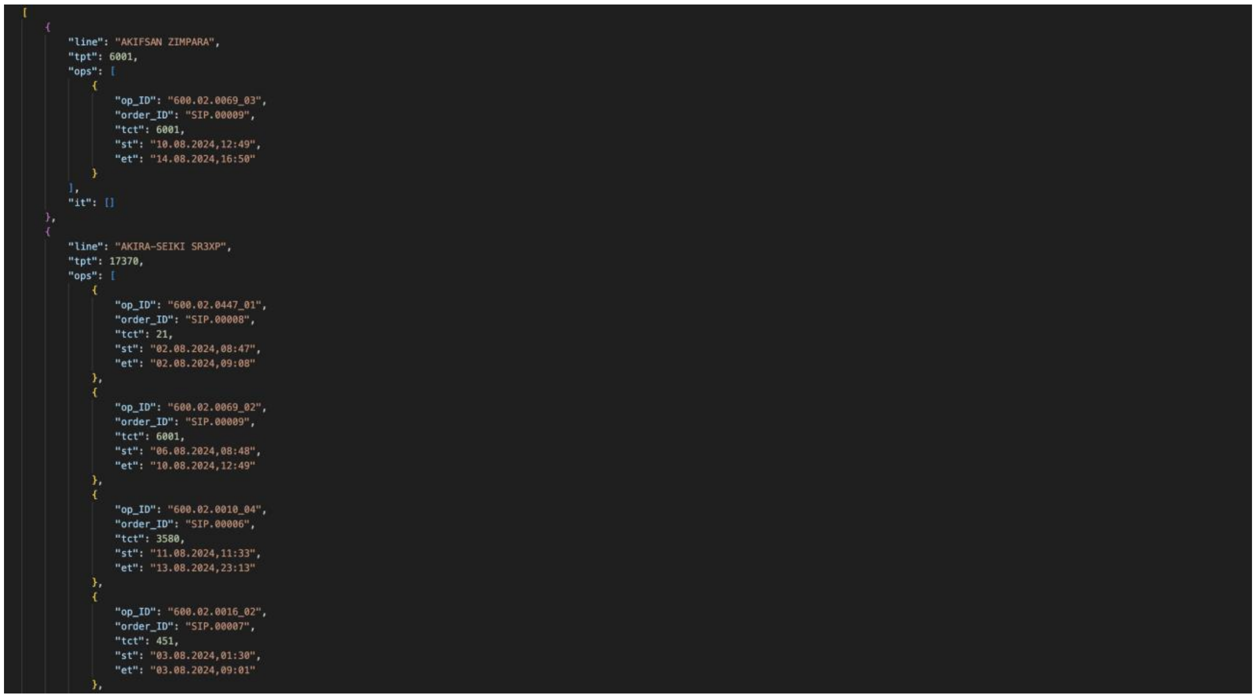


Fig. 7. Received output for *methodIndex* = 3 (machine utilization criterion), *popSize* = 400, *maxGen* = 500, *pr* = 0.03, *pc* = 1, *pm = 0.001*, *improvement_threshold* = 0.00001 and *max_stagnant_step* = 20.

Fig. 7 shows that total idle time of each machine from the current time to the end of the last operation is much less than in Fig. 6 and Fig. 4, which indicates that our algorithm is consistent. In order to reinforce that the algorithm is consistent, it was requested to run the algorithm considering the cost criterion. However, since there are many factors that trigger this objective and the only parameter available is the cost per minute of the machines, no action has been taken on this criterion at this time.

3.3.2 append.py

This script is responsible for generating JSON data containing the operations and idle times of the machines and exporting it to the file specified as 4th command line argument. In addition, the graph expressing the generational variation of the fitness value and the graph with the coexistence of the finish time and deadlines of the jobs are plotted to monitor the overall performance.



```json
[
    {
        "line": "AKIFSAN ZIMPARA",
        "tpt": 6001,
        "ops": [
            {
                "op_ID": "600.02.0069_03",
                "order_ID": "SIP.00009",
                "tct": 6001,
                "st": "10.08.2024,12:49",
                "et": "14.08.2024,16:50"
            }
        ],
        "it": []
    },
    {
        "line": "AKIRA-SEIKI SR3XP",
        "tpt": 17370,
        "ops": [
            {
                "op_ID": "600.02.0447_01",
                "order_ID": "SIP.00008",
                "tct": 21,
                "st": "02.08.2024,08:47",
                "et": "02.08.2024,09:08"
            },
            {
                "op_ID": "600.02.0069_02",
                "order_ID": "SIP.00009",
                "tct": 6001,
                "st": "06.08.2024,08:48",
                "et": "10.08.2024,12:49"
            },
            {
                "op_ID": "600.02.0010_04",
                "order_ID": "SIP.00006",
                "tct": 3580,
                "st": "11.08.2024,11:33",
                "et": "13.08.2024,23:13"
            },
            {
                "op_ID": "600.02.0016_02",
                "order_ID": "SIP.00007",
                "tct": 451,
                "st": "03.08.2024,01:30",
                "et": "03.08.2024,09:01"
            },
```

Fig. 8. Image of the output JSON file.

In Fig. 8 *'line'* represents machine, *'tpt'* represents total processing time of that machine, *'tct'* represents target cycle time of the related operation, *'st'* and *'et'* represent start and end time of the related operation respectively.

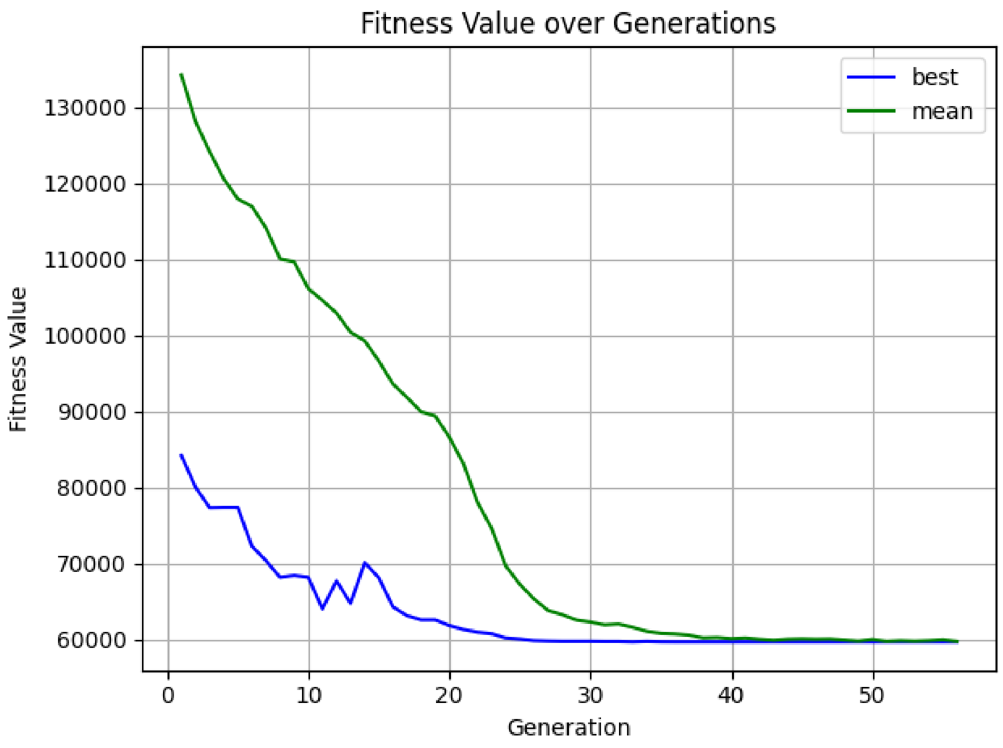


Fig. 9. Generational variation of fitness value in the maximization of machine utilization criterion.

As can be seen in Fig. 9, although the maximum number of generations (*maxGen*) is 500, the GA is terminated after 20 iterations (i.e. at generation 55) because the improvement between any two generations is less than 1 in 100000 after 35th generation. No further improvement in fitness can also be visually validated by the gradual convergence of the populations' mean fitness to the best.

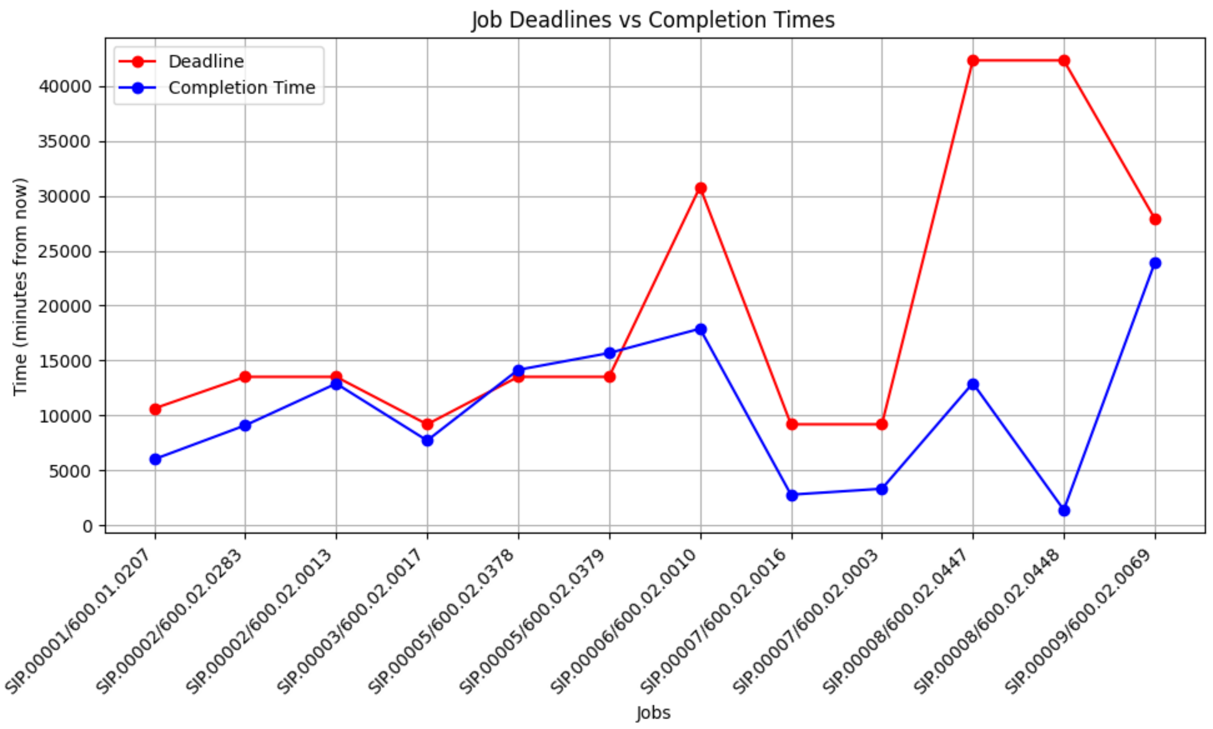


Fig. 10. The actual deadlines and finish times of the jobs to monitor overall performance.

For the graph in Fig. 10, the weighted fitness function is called and weight values of 0.35, 0.40, 0.25 are assigned to the criteria of makespan, tardiness and machine utilization, respectively, with the sum of 1. As can be observed, the blue line is generally below the red line, meaning that compliance with the tardiness criterion is at the forefront.

3.4 Further Improvements

3.4.1 Operation Partitioning

In order to further comply with the established criterions and, in fact, to produce a schedule suitable for the specific dynamics of the job shop, any operation can be split if it is not an external operation and if the total cycle time is over 720 minutes (one shift duration). Additionally, if an operation is to be split, more than one machine needs to be available.

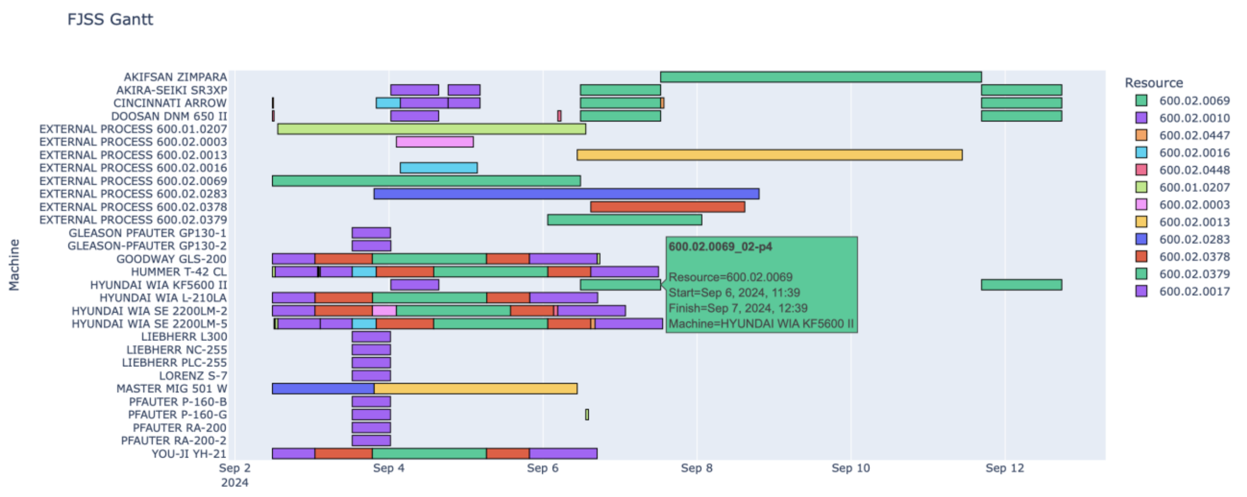


Fig. 11. Output received using operation partitioning for *methodIndex* = 4 (weighted fitness function), *popSize* = 400, *maxGen* = 500, *pr* = 0.03, *pc* = 1, *pm = 0.001*, *improvement_threshold* = 0.00001 and *max_stagnant_step* = 20.

The algorithm was executed on September 2, 2024 at 11:15 AM. The deadlines for the products given on the right side in the Fig. 11 are 22.09.2024, 23.09.2024, 02.10.2024, 09.09.2024, 02.10.2024, 10.09.2024, 09.09.2024, 12.09.2024, 12.09.2024, 12.09.2024, 12.09.2024 and 09.09.2024 respectively.

The hover text which is displayed on the operation slot in Fig. 11 represents the part of the operation of specific products. Hover text format is: *product_operation-part*.

3.4.2 Workforce Optimization

Given the relatively long cycle time of some operations, an operation may need to be split into multiple shifts. In this case, it is necessary to have more than one operator for this operation. To determine the number of operators required for a specific operation based on the unit cycle time and the number of machines, a formula is utilized that considers both the efficiency of the operator's ability to manage multiple machines and the limitations posed by the cycle time and number of machines.

The main assumptions are:

T_{cycle}	Time it takes for one machine to complete one unit of an operation.
λ	Number of shifts required to perform the operation.
$O_{capacity}$	Maximum number of machines one operator can manage effectively. This depends on the unit cycle time.
M_{max}	Threshold number of machines an operator can handle under normal circumstances. In this study, this variable determined as 3.

To determine the number of machines an operator can effectively handle, the following formula is used,

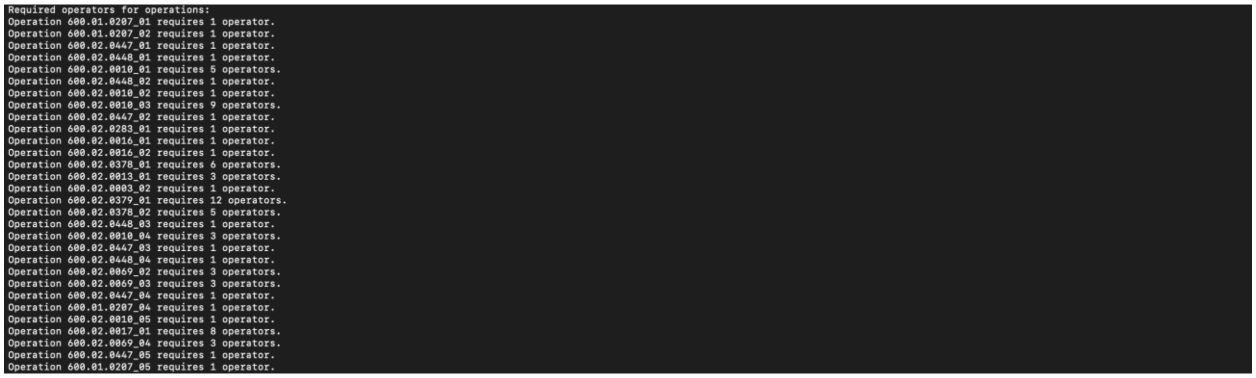
$$O_{capacity} = \frac{M_{max}}{\sqrt{T_{cycle}}},$$

where $O_{capacity}$ represents the effective machines per operator. This equation suggests that as the unit cycle time decreases (making $\sqrt{T_{cycle}}$ smaller), the effective number of machines an operator can handle increases. Conversely, with a longer unit cycle time, an operator can manage fewer machines.

To calculate the number of operators required to handle a specific number of shifts, λ, the following formula is utilized,

$$O_{required} = \left\lceil \frac{\lambda}{O_{capacity}} \right\rceil$$

Here, $O_{required}$ is the required operators, and the ceiling function $\lceil \cdot \rceil$ ensures that we round up to the nearest whole number, as partial operators are not feasible.



```
Required operators for operations:
Operation 600.01.0207_01 requires 1 operator.
Operation 600.01.0207_02 requires 1 operator.
Operation 600.02.0447_01 requires 1 operator.
Operation 600.02.0448_01 requires 1 operator.
Operation 600.02.0010_01 requires 5 operators.
Operation 600.02.0448_02 requires 1 operator.
Operation 600.02.0010_02 requires 1 operator.
Operation 600.02.0010_03 requires 9 operators.
Operation 600.02.0447_02 requires 1 operator.
Operation 600.02.0283_01 requires 1 operator.
Operation 600.02.0016_01 requires 1 operator.
Operation 600.02.0016_02 requires 1 operator.
Operation 600.02.0378_01 requires 6 operators.
Operation 600.02.0013_01 requires 3 operators.
Operation 600.02.0003_02 requires 1 operator.
Operation 600.02.0379_01 requires 12 operators.
Operation 600.02.0378_02 requires 5 operators.
Operation 600.02.0448_03 requires 1 operator.
Operation 600.02.0010_04 requires 3 operators.
Operation 600.02.0447_03 requires 1 operator.
Operation 600.02.0448_04 requires 1 operator.
Operation 600.02.0069_02 requires 3 operators.
Operation 600.02.0069_03 requires 3 operators.
Operation 600.02.0447_04 requires 1 operator.
Operation 600.01.0207_04 requires 1 operator.
Operation 600.02.0010_05 requires 1 operator.
Operation 600.02.0017_01 requires 8 operators.
Operation 600.02.0069_04 requires 3 operators.
Operation 600.02.0447_05 requires 1 operator.
Operation 600.01.0207_05 requires 1 operator.
```

Fig. 12. Console output of required operators for operations.

For instance, since the unit cycle time of CNC Lathe operation coded 600.02.0010_01 is 4 minute and the number of units to be produced is 1193, the total cycle time of this operation is 4772 minutes. In such a case, this operation is divided into 7 separate shifts of maximum 720 minutes each. Now the required operators for operation 600.02.0069_01 can be calculated as follows:

$$O_{capacity} = \frac{3}{\sqrt{4}} = 1.5,$$

$$O_{required} = \left\lceil \frac{7}{1.5} \right\rceil \approx \lceil 4.6667 \rceil.$$

Therefore operation 600.02.0010_01 requires **5** operators as can be seen in the Fig. 12.

4 Performance and Outcomes

4.1 Applying Knowledge and Skills Learned at Hacettepe

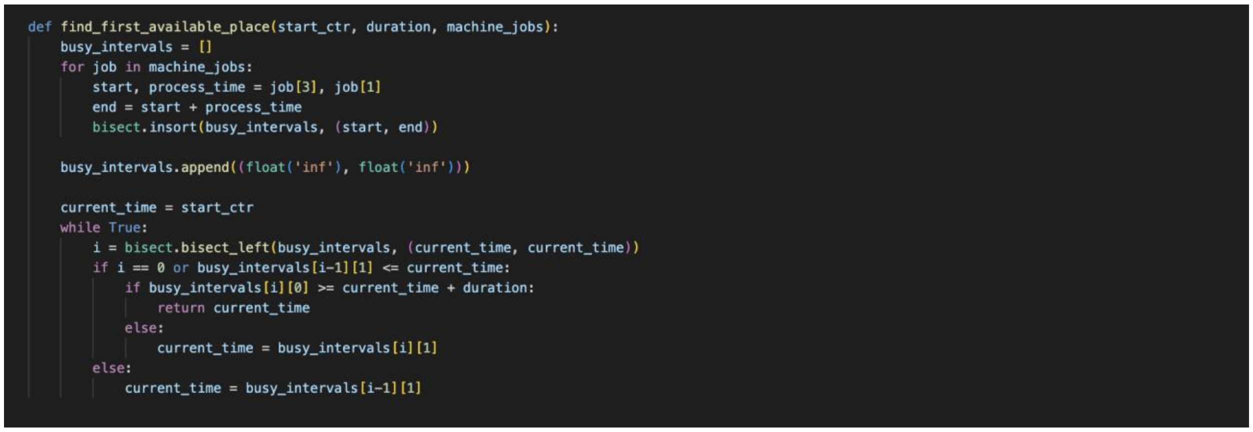
Since the code is written in Python the syntax knowledge was necessary. The Introduction to Programming (BBM101) course in the first semester was quite helpful to understand the basic syntax, data structures but especially I/O operations in Python. Also recursion was heavily utilized in the genetic operators implementations. Testing and debugging methods, which we have covered extensively in BBM101 and BBM103, are also used to handle the malfunctions.

Moreover, the program is implemented with OOP concepts that we learned in Introduction to Programming II (BBM102) to make the code more readable and modular. In this sense, inheritance and abstraction stands out relatively. Since the type of genetic operators can vary according to preferences or perhaps hardware availability, abstraction methods hide unnecessary details of the implementation and allow the user to easily modify the code through interfaces. These concepts settled in my mind better, especially with the assignments given to us in the BBM104 course.

Besides of these, the outliers in the data had to be handled so that there was no distortion in the output. In our third semester Elements of Data Science (AIN212) course, we learned a lot about how to preprocess data including outlier handling.

4.2 Solving Engineering Problems

In the initial implementation, there was a linear increase in execution time as the data size increased. This is because when trying to find a suitable location for any operation, we thoroughly check all the time intervals of the available machines for that operation.



```python
def find_first_available_place(start_ctr, duration, machine_jobs):
    busy_intervals = []
    for job in machine_jobs:
        start, process_time = job[3], job[1]
        end = start + process_time
        bisect.insort(busy_intervals, (start, end))

    busy_intervals.append((float('inf'), float('inf')))

    current_time = start_ctr
    while True:
        i = bisect.bisect_left(busy_intervals, (current_time, current_time))
        if i == 0 or busy_intervals[i-1][1] <= current_time:
            if busy_intervals[i][0] >= current_time + duration:
                return current_time
            else:
                current_time = busy_intervals[i][1]
        else:
            current_time = busy_intervals[i-1][1]
```

Fig. 13. Method to find the first available place for the operation slots.

In order to develop a solution to this, the `bisect` module is used for efficiently managing a sorted list of machine time availability and operation completion times. The primary use case is to keep the list ordered and find the appropriate position in the list to add a new element.

When scheduling jobs on machines, the end times of operations need to be tracked to determine when a machine will be available next. The `bisect.insort` function inserts an operation's end time into a sorted list of existing end times to ensure the list stays in sorted order.

Additionally, when assigning operations, it is necessary to find the first available slot for an operation on a given machine. `bisect_left(busy_intervals, (current_time, current_time))` returns the index `i` where the tuple `(current_time, current_time)` can be inserted in `busy_intervals` while maintaining the order. This operation has a time complexity of O(*log n*), where *n* is the number of busy intervals.

`bisect_left` result is then utilized to determine whether the `current_time` can be used to start a new operation without overlap the existing ones. This decision making is done by checking the relationship between `current_time` and the end times of the previous/current intervals as can be seen in Fig. 13.

Which engineering problems did you solve related to the computer systems and applications during your internship? Explain in detail.

4.3 Teamwork

If we talk about team dynamics briefly, after the requirements in any project are specified by the manager, everyone works in feedback with each other to complete their part of the task and the decisions are made after consulting the manager. Team members informations who I worked with are given below:

(1) Hayri Kılıç
After working as an intern engineer in five different companies in his field (Electrical/Electronic, PLC Automation, Embedded Software, Hardware Design, etc.), he worked as a freelancer for a while. As of 2021, he is employed as an "Embedded Software/Hardware Design Engineer" at the *upu.io* R&D center of Baca Engineering Software Defense LTD. Co. to date, he has received training and certificates in Device & System Development, EMC Design Training, Hardware Design Engineering with Altium Designer, Python 3, Computer Security 2016, Alternative Energy Sources 2017, Wireless Sensors 2018, and Strategic Financial Leadership: CFOs of the Future 2018. He has also completed the Quality Systems, ISO 9001:2015 Standards training in 2019. He has been a member of the Chamber of Electrical Engineers (EMO) since 2018. He holds a language education and certification at an intermediate level (B1) in English.

(2) Göktürk Çelebioğlu
He has a background in industrial engineering with experience in both the software and manufacturing industries. He completed his a Bachelor’s and Master’s degree in Industrial Engineering at Başkent University, where he studied machine learning, scheduling, linear programming, and decision-making systems. Initially he worked as a Business Analyst at *Arma Group Holding*, managing product features and planning tasks for cloud storage and email products. Later, he worked as an Industrial Engineer at *Pars Savunma & Havacılık* and as a Planning Specialist at *Armtek Elektrik*, handling production and material planning for transformers. Now, he work as an Industrial Engineer at *upu.io*. He also have certifications in machine learning and Lean Six Sigma.

I worked with Göktürk Çelebioğlu on the dynamics of the job shop in general; appropriate data about machines, jobs, operations and analyzing this data. Also, all the necessary formulas were calculated together.

With Hayri Kılıç, we worked on implementation details, code architecture and necessary improvements, as well as how to store data, output format and how to export data.

4.4 Multi-Disciplinary Work

In order to bring the data into proper form, the factory's work strategy had to be determined. For this, machine, order and operation information was analyzed with existing industrial engineers. In addition, ideas were exchanged on how to visualize the planned operations and it was decided that Gantt chart representation was the most appropriate.

In the early stages of the implementation, the objective functions that determine the fitness value could only be used separately, but with the ongoing discussions and developments, all the objectives were included in the fitness function and their importance was determined by the weight values assigned to these objectives as mentioned in ‘Selection’ part of **3.1.2**.

Also, a superficial formulation explained in section **3.3.2** was developed with industrial engineers to determine the number of operators required to perform any operation in workforce optimization.

4.5 Professional and Ethical Issues

One problem mentioned to me is that operators are worried about losing their jobs due to the digitalization of the workshop and feel that they are incompetent in their work. But in fact, rather than putting operators out of business, this digitalization aims to create a digital twin of the workplace, making activities such as management and monitoring much more easier.

On the other hand, this digitalization is necessary to measure the operators' effectiveness. Not surprisingly, nobody wants to retain staff who are not performing adequately in their jobs. In this context, in order to motivate operators more for their work, a bonus pool worth ₺30k was created where they can receive bonuses based on their monthly performance.

4.6 Impact of Engineering Solutions

Economically speaking, the “Plug and Play” approach significantly reduces installation complexity and eliminates the need for intensive engineering and cabling processes. This results in lower initial costs with an 85% reduction in investment compared to its counterparts.

From an environmental perspective, *upu.io*'s Carbon Footprint Calculator (*upu.c-count*) provides a tangible way for companies to track and reduce their carbon emissions. This fosters environmentally responsible behavior by helping companies understand their product-level environmental impact and make informed decisions to enhance sustainability.

On a global and societal level, *upu.io*’s streamlined solutions enable companies to adopt advanced technology without the high financial and technical barriers typically associated with it. This is not only democratizing access to innovation but also promoting sustainability and efficiency across industries etc.

4.7 Locating Sources and Self-Learning

When developing this program, I mainly used papers and journal articles that were available on the internet, some of which I inspected on the recommendation of the people I worked with in the team.

Although the general working principle of the GA is almost same everywhere, it is retrieved from Zhang et al. [2012]. Encoding logic is derived from Gao et al. [2006] in this study. Moreover, chromosome representation is adopted from Ho et al. [2007] which evaluates chromosome according to two separate subproblems. In addition, general mathematical models and notations of FJSP can refer to Ozguven et al. [2010] and Roshanaei et al. [2013]. Some of the utilized crossover methods are received from Ripon et al. [2011] and Ono et al. [1996]

I also found implementation examples for our selected/candidate selection, crossover and mutation operator types (JBX, POX, neighborhood mutation, swapping mutation etc.) in a few repositories on GitHub and it actually helped me a lot in the development process of genetic operators.

4.8 Using New Tools and Technologies

We tried AWS IoT Core's MQTT client component to transmit the obtained Gantt chart and output data in JSON format to a specific endpoint/topic after the scheduling request is received.

After the code is executed, it waits for JSON input to be published to the topic specified in the implementation. Once the message is received, the GA runs and the content of the JSON file containing the parameters of the optimal solution is published back to the topic.

I carried out the implementation in consultation with my supervisor as he is experienced in this field. Even though there is no UI/UX interface to make it easy to use, this gave me an important knowledge about AWS IoT Core applications and coding in Cloud9 IDE.

5 Conclusions

Adaptability is particularly crucial in the production field, where demand, resource availability, and operational conditions can fluctuate. Therefore, the application of AI and GAs in such environments are stimulating innovation, reducing inefficiencies, leading to more effective use of resources etc.

In that sense, I developed an efficient production scheduling and workforce optimization algorithm that aims to minimize the fitness values of the criteria determined for FJSP by using the population-based evolutionary search capability of GA. Six different order data of the company were used to test the performance of the GA. Experimental results show that the proposed GA provides a significant improvement for solving FJSP regardless of solution accuracy and computation time.

In this process, the knowledge and skills I learned at Hacettepe, especially in programming, were key factors. When I combine the skills I gained with the necessary resources, I can say that I had a truly important internship process. I also gained important experience not only in technical terms but also in business ethics and order.

References

[1] Gao, L., Peng, C.Y., Zhou, C., Li, P.G., 2006. Solving exible job shop scheduling problem using general particle swarm optimization. In: Proceedings of the 36th CIE Conference on Computers and Industrial Engineering, pp. 3018–3027.

[2] Ho, N.B., Tay, J.C., Lai, E.M.K., 2007. An effective architecture for learning and evolving flexible job shop schedules. Eur. J. Oper. Res. 179, 316–333.

[3] Ozguven, C., Ozbakir, L., Yavuz, Y., 2010. Mathematical models for job shop scheduling problems with routing and process plan exibility. Appl. Math. Model. 34, 1539–1548.

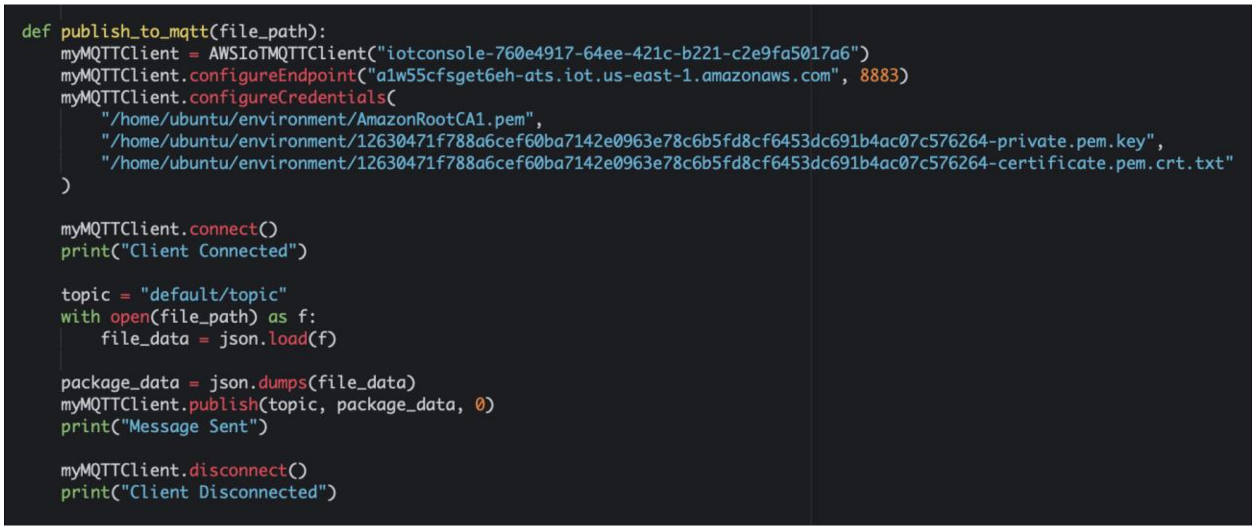
[4] Roshanaei, V., Azab, A., ElMaraghy, H., 2013. Mathematical modelling and a metaheuristic for flexible job shop scheduling. Int. J. Prod. Res. 51 (20), 6247–6274.

[5] Zhang, Q., Manier, H., Manier, M.A., 2012. A genetic algorithm with tabu search procedure for exible job shop scheduling with transportation constraints and bounded processing times. Comput. Oper. Res. 39, 1713–1723.

[6] Ripon, K.S.N., Siddique, N.H. & Torresen, J., 2011. Improved precedence preservation crossover for multi-objective job shop scheduling problem. Evolving Systems 2, 119–129

[7] Ono I., Yamamura M., Kobayashi S., 1996. "A genetic algorithm for job-shop scheduling problems using job-based order crossover," Proceedings of IEEE International Conference on Evolutionary Computation, Nagoya, Japan, pp. 547-552

Appendices



```
def publish_to_mqtt(file_path):
    myMQTTClient = AWSIoTMQTTClient("iotconsole-760e4917-64ee-421c-b221-c2e9fa5017a6")
    myMQTTClient.configureEndpoint("a1w55cfsget6eh-ats.iot.us-east-1.amazonaws.com", 8883)
    myMQTTClient.configureCredentials(
        "/home/ubuntu/environment/AmazonRootCA1.pem",
        "/home/ubuntu/environment/12630471f788a6cef60ba7142e0963e78c6b5fd8cf6453dc691b4ac07c576264-private.pem.key",
        "/home/ubuntu/environment/12630471f788a6cef60ba7142e0963e78c6b5fd8cf6453dc691b4ac07c576264-certificate.pem.crt.txt"
    )

    myMQTTClient.connect()
    print("Client Connected")

    topic = "default/topic"
    with open(file_path) as f:
        file_data = json.load(f)

    package_data = json.dumps(file_data)
    myMQTTClient.publish(topic, package_data, 0)
    print("Message Sent")

    myMQTTClient.disconnect()
    print("Client Disconnected")
```

Fig. 14. AWS MQTT client implementation in Cloud9 IDE.