Faculty of Engineering and Technology

Electrical and Computer Engineering Department

Artificial Intelligence – ENCS3340

Project 1

Optimizing Job Shop Scheduling in a Manufacturing Plant using Genetic Algorithm

---

Group Members:

Doaa Hatu 1211088

Mai Beitnoba 1210260

Instructor:

Yazan AbuFarha

Section: 4

Date: 18/05/2024

Abstract

This project addresses the Job Shop Scheduling Problem (JSSP) using a Genetic Algorithm (GA) to optimize job sequences on multiple machines, aiming to minimize the makespan. By encoding potential schedules as chromosomes and evolving them through selection, crossover, and mutation, the GA iteratively improves solutions based on a fitness function that evaluates makespan. The process starts with initializing job details and generating an initial population of schedules. Through successive generations, the GA converges on an efficient schedule, which is then visualized using a Gantt chart. The results demonstrate the effectiveness of GAs in solving complex scheduling problems, offering a practical solution for enhancing productivity in industrial operations.

Contents

Abstract 2

Table of Figures 4

Genetic algorithm 5

Problem Formulation 6

- **Population** 6
- **Chromosome Representation** 6
- **Fitness Function** 6
- **Selection** 6
- **Crossover** 6
- **Mutation** 6
- **Termination Criteria** 6

Test cases 7

- **Test case 1:** 7
- **Test case 2:** 9
- **Test case 3:** 11

Conclusion 13

Appendix 14

Table of Figures

Figure 1: Test Case 1 results 7
Figure 2: Test Case 1 plot 8
Figure 3: Test Case 2 results 9
Figure 4: Test Case 2 plot 10
Figure 5: Test Case 3 results 11
Figure 6: Test Case 3 plot 12

Genetic algorithm

Genetic Algorithms (GAs) are adaptive heuristic search methods that fall under the broader category of evolutionary algorithms. Rooted in the concepts of natural selection and genetics, these algorithms intelligently exploit random searches by using historical data to steer the search towards regions of better performance within the solution space. They are widely employed for generating high-quality solutions to optimization and search problems.

Genetic algorithms mimic the process of natural selection, where species that adapt well to environmental changes survive, reproduce, and pass on their traits to the next generation. Essentially, they emulate the "survival of the fittest" principle among individuals across successive generations to solve problems. Each generation comprises a population of individuals, with each individual representing a potential solution or a point in the search space. These individuals are typically encoded as strings of characters, integers, floats, or bits, similar to chromosomes in biology.

Problem Formulation

Population

It is a collection of individual solutions (chromosomes), in this project it consists of multiple schedules for different machines.

Chromosome Representation

It represents a single solution, in the project, each chromosome is a schedule (includes order of jobs and timing of operations for jobs in the machine).
In our genetic algorithm for the Job Shop Scheduling Problem (JSSP), each chromosome is represented as a sequence of job IDs, with each job ID corresponding to an operation of that job. A chromosome is essentially a list where the position and value of each element denote the operation sequence and job ID, respectively. For instance, the chromosome [1, 1, 1, 2, 2, 4, 3, 3, 4, 1, 2] 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 4 Operation 1, Job 3 Operation 1, Job 3 Operation 2, Job 4 Operation 2, Job 1 Operation 4, and Job 2 Operation 3. 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. By preserving the sequence integrity and ensuring that each job's operations appear the appropriate number of times, the algorithm can effectively explore the solution space and evolve towards an optimal schedule.

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).

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 (specified by user).

Test cases

Test case 1:

File content

Job_1: M1[10] -> M2[5] -> M4[12]
Job_2: M2[7] -> M3[15] -> M6[8]
Job_3: M1[7] -> M3[10] -> M7[5]
Job_4: M5[7] -> M4[13] -> M3[3] -> M5[5]
Job_5: M3[7] -> M2[5] -> M2[7] -> M2[10]
Job_6: M2[7] -> M6[8] -> M1[17]
Job_7: M6[7] -> M7[10] -> M1[8]

Results

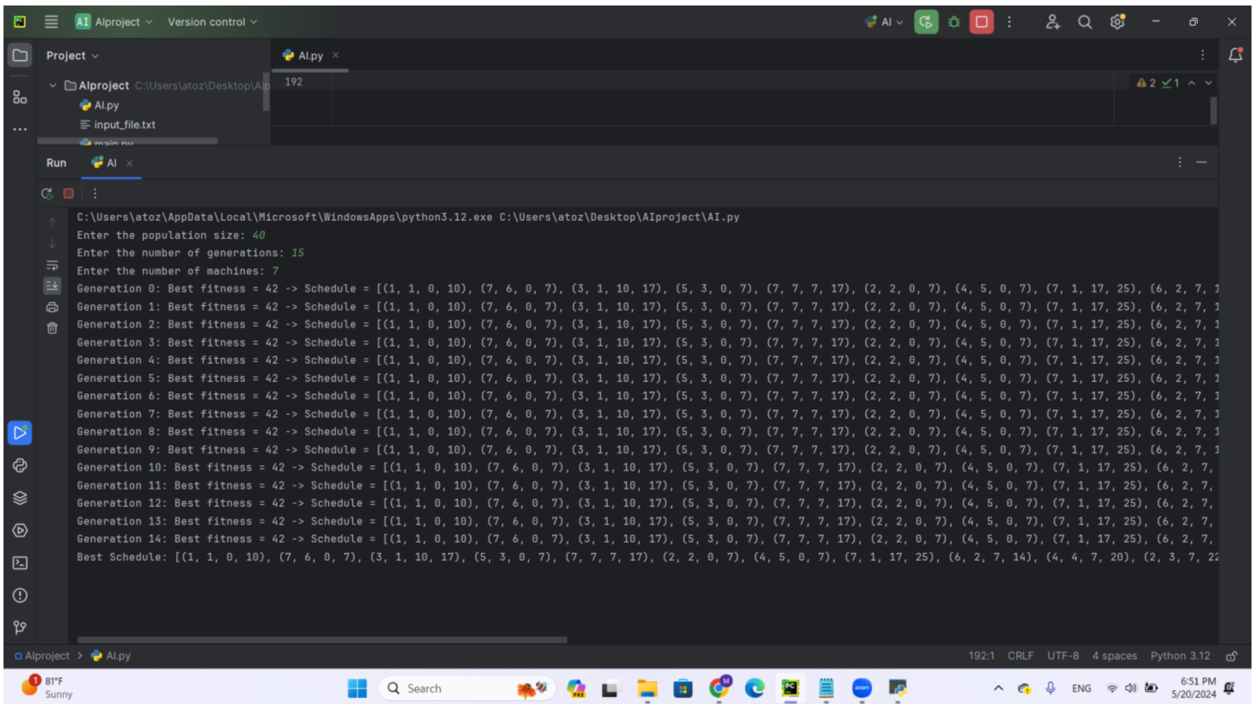

Figure 1: Test Case 1 results

Plot

Figure 2: Test Case 1 plot

Test case 2:

File content

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

Results

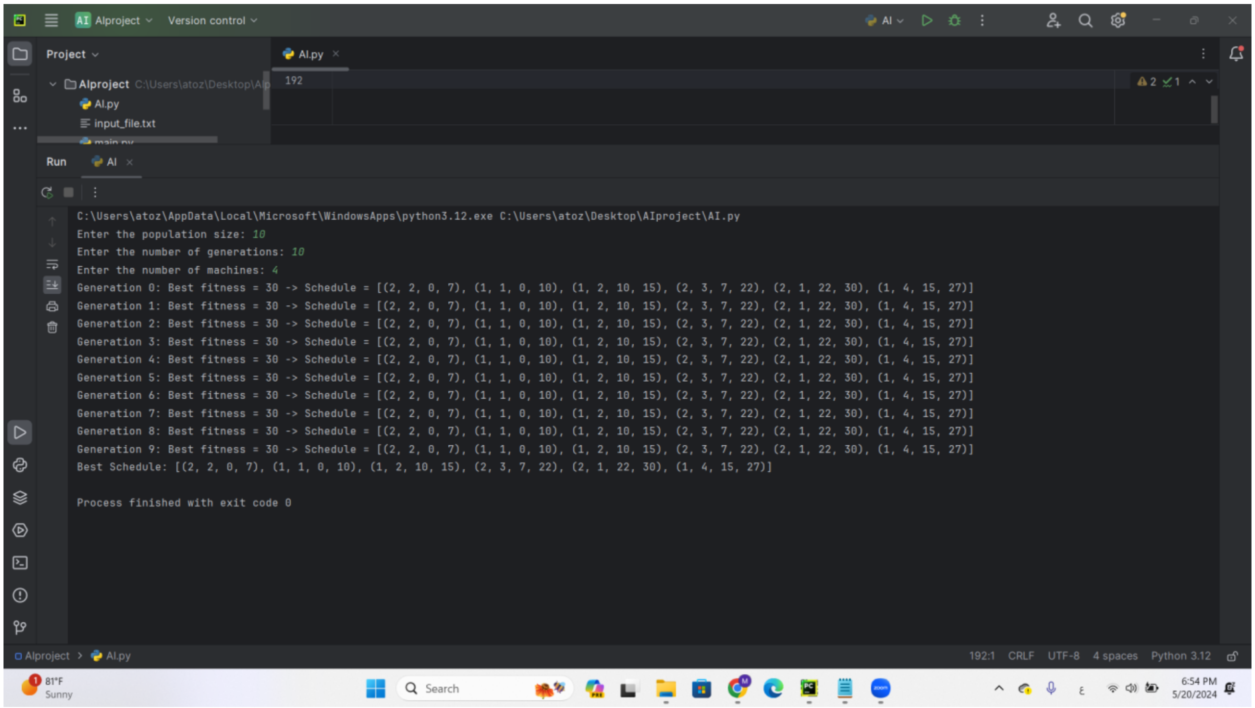

Figure 3: Test Case 2 results

Plot

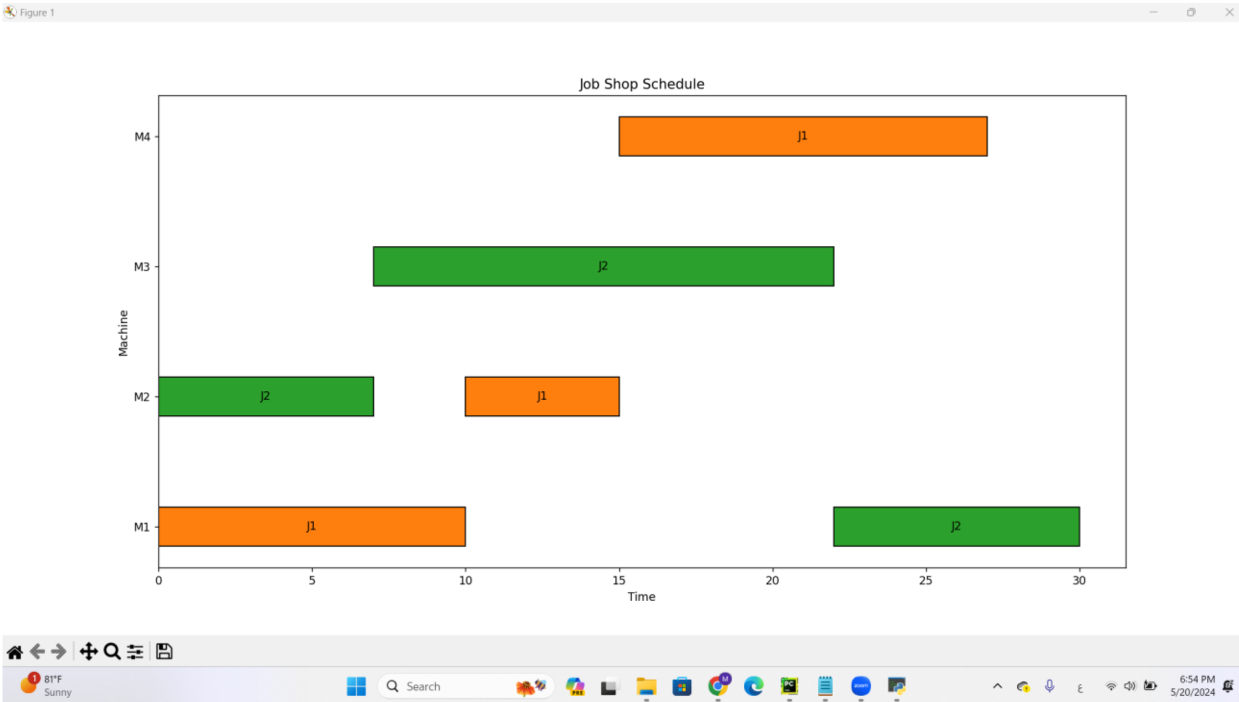

Figure 4: Test Case 2 plot

Test case 3:

File content

Job_1: M1[7] -> M4[13] -> M3[5]

Job_2: M2[7] -> M1[5]

Job_3: M1[10] -> M2[5] -> M4[12]

Job_4: M2[7] -> M3[15] -> M1[8]

Results

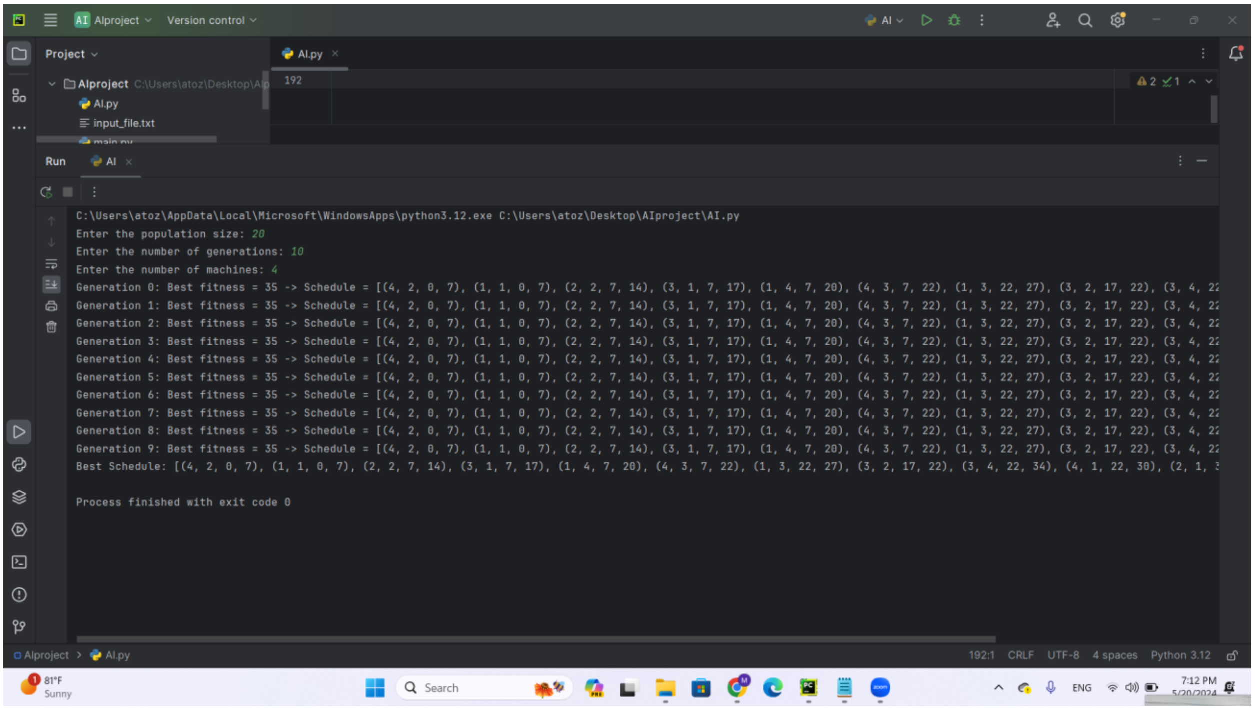

Figure 5: Test Case 3 results

Plot

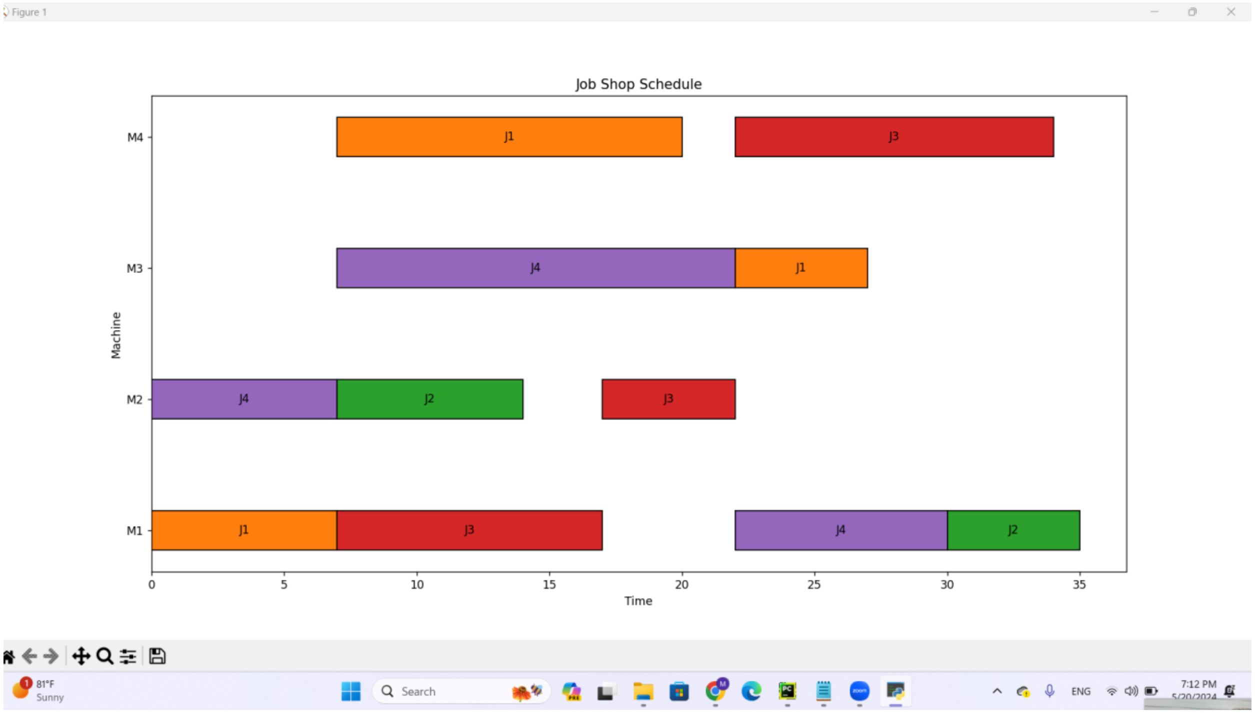

Figure 6: Test Case 3 plot

Conclusion

In conclusion, the application of a Genetic Algorithm to the Job Shop Scheduling Problem has proven to be an effective method for optimizing the allocation of jobs across multiple machines. By leveraging evolutionary principles, the algorithm successfully minimized the makespan, demonstrating its capability to handle complex scheduling scenarios. The iterative process of selection, crossover, and mutation allowed the algorithm to explore a wide solution space and converge on a highly efficient schedule. The visualization of the results through Gantt charts provided clear insights into the optimized job sequences and machine utilization. This approach not only enhances operational efficiency but also offers a scalable solution adaptable to various industrial contexts, underscoring the practical value of genetic algorithms in solving real-world scheduling challenges.

Appendix

```
import random
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

mutation_rate = 0.01
selection_rate = 0.2

Population_Size = int(input("Enter the population size: "))
Number_of_Generations = int(input("Enter the number of generations: "))
num_machines = int(input("Enter the number of machines: "))
list_of_jobs = []


def initialize_jobs_from_file(file_name):
    with open(file_name, 'r') as file:
        lines = file.readlines()

    for line in lines:
        job_id = int(line.split(":")[0].split("_")[1])
        data = line.split(":")[1].strip().split("->")
        num_operations = 0
        operations = []
        for op_data in data:
            machine, processing_time = int(op_data.split("M")[1].split("[")[0]),
int(
                op_data.split("[")[1].split("]")[0])
            operations.append({'machine': machine, 'processing_time':
processing_time})
            num_operations += 1
        list_of_jobs.append({'job_id': job_id, 'num_operations': num_operations,
'operations': operations})
    # print(list_of_jobs)


def initialize_jobs():
    num_jobs = int(input("Enter the number of jobs: "))
    for i in range(num_jobs):
        job_id = i + 1
        num_operations = int(input(f"Enter the number of operations for Job {i +
1}: "))
        operations = []
        for j in range(num_operations):
            while True:
```

```
                machine = int(input(f"Enter machine for operation {j + 1} of Job
{i + 1}: "))
                if machine > num_machines:
                    print(f"Invalid machine number. Please enter a valid machine
number (1 to {num_machines}).")
                else:
                    break
            processing_time = int(input(f"Enter processing time for operation {j
+ 1} of Job {i + 1}: "))
            operations.append({'machine': machine, 'processing_time':
processing_time})

        list_of_jobs.append({'job_id': job_id, 'num_operations': num_operations,
'operations': operations})


def initialize_chromosome():
    chromosome = []
    for job in list_of_jobs:
        job_id = job['job_id']
        for operation in range(job['num_operations']):
            chromosome.append(job_id)
    return chromosome


def initialize_population():
    initial_chromosome = initialize_chromosome()
    population = []
    for i in range(Population_Size):
        chromosome_copy = initial_chromosome[:]
        random.shuffle(chromosome_copy)
        population.append(chromosome_copy)
    return population


def fitness_func(chromosome):
    machine_avail_time = {machine: 0 for machine in range(1, num_machines + 1)}
    job_completion_time = {job['job_id']: 0 for job in list_of_jobs}
    job_operation_index = {job['job_id']: 0 for job in list_of_jobs}
    schedule = []

    for job_id in chromosome:
        job = list_of_jobs[job_id - 1]
        operations = job['operations']
        op_index = job_operation_index[job_id]
```

```
        # Ensure the operation index is within the bounds
        if op_index >= len(operations):
            continue

        operation = operations[op_index]
        machine = operation['machine']
        processing_time = operation['processing_time']
        start_time = max(machine_avail_time[machine],
job_completion_time[job_id])
        completion_time = start_time + processing_time
        machine_avail_time[machine] = completion_time
        job_completion_time[job_id] = completion_time
        job_operation_index[job_id] += 1

        schedule.append((job_id, machine, start_time, completion_time))

    makespan = max(job_completion_time.values())
    return makespan, schedule


def select_parents(population):
    parents = []
    for _ in range(2):
        tournament = random.sample(population, k=3)
        parents.append(min(tournament, key=lambda x: fitness_func(x)[0]))
    return parents


def crossover(parent1, parent2):
    point = random.randint(1, len(parent1) - 1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]

    child1 = validate_and_repair(child1)
    child2 = validate_and_repair(child2)

    return child1, child2


def validate_and_repair(child):
    job_operation_counts = {job['job_id']: job['num_operations'] for job in
list_of_jobs}
    child_counts = {job_id: child.count(job_id) for job_id in
job_operation_counts}
```

```
    for job_id, expected_count in job_operation_counts.items():
        if child_counts[job_id] != expected_count:
            current_count = child_counts[job_id]
            if current_count < expected_count:
                missing_count = expected_count - current_count
                for _ in range(missing_count):
                    child.append(job_id)
            elif current_count > expected_count:
                excess_count = current_count - expected_count
                indices_to_remove = [i for i, x in enumerate(child) if x ==
job_id][:excess_count]
                for index in sorted(indices_to_remove, reverse=True):
                    child.pop(index)

    return child


def mutation(chromosome):
    if random.random() < mutation_rate:
        i, j = random.sample(range(len(chromosome)), 2)
        chromosome[i], chromosome[j] = chromosome[j], chromosome[i]


def genetic_algorithm():
    population = initialize_population()
    best_schedule = None
    for generation in range(Number_of_Generations):
        new_population = []
        for _ in range(Population_Size // 2):
            parents = select_parents(population)
            child1, child2 = crossover(parents[0], parents[1])
            mutation(child1)
            mutation(child2)
            new_population.extend([child1, child2])

        # population = sorted(new_population, key=lambda x:
fitness_func(x)[0])[:Population_Size]
        best_schedule = min(population, key=lambda x: fitness_func(x)[0])
        BF, BS = fitness_func(best_schedule)
        print(f"Generation {generation}: Best fitness = {BF} -> Schedule = {BS}")

    return best_schedule
```

```
def plot_gantt_chart(schedule):
    fig, ax = plt.subplots()
    cmap = ListedColormap(
        ['tab:blue', 'tab:orange', 'tab:green', 'tab:red', 'tab:purple',
'tab:brown', 'tab:pink', 'tab:gray',
         'tab:olive', 'tab:cyan'])

    for job_id, machine, start_time, end_time in schedule:
        ax.barh(machine, end_time - start_time, left=start_time, height=0.3,
color=cmap(job_id % 10), edgecolor='black')
        ax.text(start_time + (end_time - start_time) / 2, machine, f'J{job_id}',
color='black', ha='center',
                va='center')

    ax.set_xlabel('Time')
    ax.set_ylabel('Machine')
    ax.set_title('Job Shop Schedule')
    plt.yticks(range(1, num_machines + 1), [f'M{i}' for i in range(1,
num_machines + 1)])
    plt.show()


# initialize_jobs()
initialize_jobs_from_file("input_file.txt")
Population = initialize_population()

best_chromosome = genetic_algorithm()
_, best_schedule = fitness_func(best_chromosome)
print("Best Schedule:", best_schedule)

plot_gantt_chart(best_schedule)
```