Universitat Rovira i Virgili

Delivery of Activity 2

Neural and Evolutionary Computation

Optimization with Genetic Algorithms

Javier Novella Ruiz

Contents

1 Introduction

The Genetic Algorithms (GA) [1] are part of what it is called Evolutionary Computation [2].

In evolutionary computation, an initial set of candidate solutions is generated and iteratively updated. Each new generation is produced by stochastically removing less desired solutions, and introducing small random changes as well as, depending on the method, mixing parental information.

Genetic Algorithms (GA) are metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA), as mentioned before. Genetic algorithms are used to generate solutions to optimization and search problems via biologically inspired operators such as selection, crossover, and mutation.

In this work a Genetic Algorithm has been implemented to find solutions to the Job-Shop Scheduling Problem (JSSP) [3].

1.1 Technology used

The following technology has been used for the development:

- Visual Studio Code v1.96.4
- Python v3.13.0
- Jupyter Notebooks with Jupyter VSCode extension v2024.11.0

2 Technical background

2.1 Genetic algorithm

Genetic Algorithms (GA) are inspired by Darwinian evolution and genetic principles. Each element in the search space (1) is represented by a chromosome (2). The search space consists of individuals, represented as vectors of 0s and 1s, with each component being a gene.

$$S = \{c \in \{0,1\}^m\} \tag{1}$$

$$c \in \{0,1\}^m \tag{2}$$

The process starts with an initial population of chromosomes. The population evolves through selection, reproduction, and mutation. Natural selection favors individuals that are best suited for the environment, driving optimization. The fitness (3) of each individual determines its ability to survive and reproduce, influencing the next generation in the population (4).

$$F(c) \geq 0 \tag{3}$$

$$P = \{c^\mu\}_{\mu=1,\ldots,p} \subset S \tag{4}$$

2.1.1 Selection

Selection [4] is the process of choosing individuals from the population based on their fitness. The better the fitness, the higher the chance of being selected for reproduction. This simulates natural selection, where the fittest individuals are more likely to pass on their genes to the next generation.

Roulette wheel
Roulette wheel selection is a probabilistic method where individuals in the population are selected based on their fitness. The selection probability is proportional to the fitness value, with fitter individuals having a higher chance of being selected.

Tournament selection
tournament selection involves selecting a subset of individuals at random and choosing the best among them to be part of the next generation. This method introduces competition and increases the selection pressure on fitter individuals.

2.1.2 Crossover

Crossover [5] is the process of combining the genetic material of two parent individuals to create offspring. It involves exchanging parts of their chromosomes to generate new solutions. This mimics the process of sexual reproduction in nature and helps combine traits from both parents.

One-point
One-point crossover is a genetic algorithm technique where two parent solutions are combined to create offspring. A crossover point is selected at random, and the genetic material from the parents is swapped at this point to produce two new solutions.

Uniform
Uniform crossover, however, combines the parents by selecting genes from each parent at each position based on a probability. This results in offspring that have a more mixed combination of their parents' traits, maintaining diversity within the population.

2.1.3 Mutation

Mutation [6] introduces small random changes to an individual's chromosome. It helps maintain genetic diversity within the population and prevents premature convergence. Mutation allows the algorithm to explore new areas of the solution space that might not be reachable through selection and crossover alone.

One-gene
One-gene mutation refers to altering a single gene in an individual's chromosome. This mutation is typically performed by flipping a bit in the gene sequence (from 0 to 1 or vice versa), which introduces small variations in the population.

Pobabilistic
Probabilistic mutation, on the other hand, alters genes based on a certain probability. The probability of mutation may vary, and it is typically applied to a all set of genes, ensuring a controlled but diverse exploration of the solution space.

2.2 Job-Shop Schedule Problem (JSSP)

The Job-Shop Scheduling Problem (JSSP) involves assigning a set of jobs $J = \{J_1, J_2, ..., J_n\}$ to a set of machines $M = \{M_1, M_2, ..., M_n\}$ while respecting specific constraints. Each job consists of a sequence of operations $O = \{O_1, O_2, ..., O_n\}$ that need to be processed on certain machines, with each operation requiring a specified amount of time. The goal is to determine an optimal schedule that minimizes the makespan, which is the total time required to complete all jobs, while ensuring that machine constraints and job sequences are adhered to.

Suppose also that there is some cost function $C : X \to [0, +\infty]$. The cost function may be interpreted as a "total processing time", and may have some expression in terms of times $C_{ij} : M \times J \to [0, +\infty]$, the cost/time for machine M_i to do job J_j.

The job-shop problem is to find an assignment of jobs $x \in X$ such that $C(x)$ is a minimum, that is, there is no $y \in X$ such taht $C(x) > C(y)$.

3 Implementation

To implement the genetic algorithm for the Job-Shop Scheduling Problem (JSSP), a Python script in Jupyter Notebooks has been created. For better understanding of the algorithm developed, it is going to be broken down and explained step by step.

3.1 Software architecture - GeneticOptimizatorJSSP class

To develop the solution a new class has been developed GeneticOptimizatorJSSP. This class encapsulates the logic of the genetic algorithm for solving the Job-Shop Scheduling Problem, making the implementation modular and reusable. By organizing the methods and data within a class, the code becomes more maintainable, as the different components of the algorithm (population initialization, fitness evaluation, selection, crossover, and mutation) are neatly separated and can be modified independently. It also allows for easy instantiation and reuse of the optimization process, enabling the user to work with different sets of job data or modify parameters without altering the core logic. Using a class structure enhances readability, scalability, and debugging, making it more efficient to expand or adjust the algorithm in the future.

3.1.1 Methods

- __init__(self): Initializes the class with empty job data, solution lists, and best solution placeholders.
- _get_number_of_machines(self, jobs): Calculates the number of machines used in the job shop based on the job data.
- get_jobs(self): Returns the current job data.
- set_jobs_from_string(self, jobs): Sets job data from a string, converting it into a list of operations and machines.
- set_jobs_from_file(self, jobs): Sets job data from a file, reading and parsing the operations and machines.
- _initialize_population(self, population_size): Generates an initial population of chromosomes (solutions) by creating random permutations of job operations, respecting job dependencies.

- _evaluate_fitness(self, population): Calculates the fitness of each individual in the population based on the makespan, or total processing time.
- _selection(self, population, fitness, method='roulette-wheel'): Selects two parents from the population using either roulette-wheel or tournament selection methods.
- _crossover(self, individual_1, individual_2, method='one-point'): Performs crossover on two selected individuals to produce two offspring using either uniform or one-point method.
- _mutate(self, individual, method, probability=0.5): Applies mutation on an individual with a specified probability, either flipping one gene or applying probabilistic mutation.
- _elitism(self, popuplation_1, population_1_fitness, population_2, number=1): Ensures the best individuals from the previous generation are retained in the new population.
- _get_best_solution(self, population, fitness): Finds the best solution in the population by identifying the individual with the lowest makespan.
- run_genetic_algorithm(self, num_generations, population_size, selection_method, crossover _method, mutation_method, mutation_probability, elitism_number): Runs the genetic algorithm for a specified number of generations, performing selection, crossover, mutation, and elitism to optimize the job-shop schedule.

3.1.2 Attributes

- jobs: Stores the job data, which includes operations and processing times for each job.
- number_of_jobs: Holds the total number of jobs in the job-shop scheduling problem.
- number_of_machines: Stores the number of distinct machines used in the job-shop scheduling.
- solutions: A list that holds all the solutions (chromosomes) in the population.
- solutions_fitness: Holds the fitness values (makespan values) for each solution in the population.
- best_solution: Stores the best solution (chromosome) found during the genetic algorithm execution.
- best_solution_fitness: Holds the fitness (makespan) value of the best solution found.

3.2 Loading the problem

The problem is loaded using the methods set_jobs_from_string(self, jobs), that sets job data from a string, or set_jobs_from_file(self, jobs), that eets job data from a file. independently of the method used the expected format of the data is a matrix ($n_{jobs} \times n_{machines}$) each element of the matrix is formed by the pair machine number and processing time m_i, t_i. Here it is an example from https://people.brunel.ac.uk/ mastjjb/jeb/orlib/files/jobshop1.txt, specifically

the Adams, Balas, and Zawack 10×10 instance.

$$\begin{bmatrix}
4\ 88 & 8\ 68 & 6\ 94 & 5\ 99 & 1\ 67 & 2\ 89 & 9\ 77 & 7\ 99 & 0\ 86 & 3\ 92 \\
5\ 72 & 3\ 50 & 6\ 69 & 4\ 75 & 2\ 94 & 8\ 66 & 0\ 92 & 1\ 82 & 7\ 94 & 9\ 63 \\
9\ 83 & 8\ 61 & 0\ 83 & 1\ 65 & 6\ 64 & 5\ 85 & 7\ 78 & 4\ 85 & 2\ 55 & 3\ 77 \\
7\ 94 & 2\ 68 & 1\ 61 & 4\ 99 & 3\ 54 & 6\ 75 & 5\ 66 & 0\ 76 & 9\ 63 & 8\ 67 \\
3\ 69 & 4\ 88 & 9\ 82 & 8\ 95 & 0\ 99 & 2\ 67 & 6\ 95 & 5\ 68 & 7\ 67 & 1\ 86 \\
1\ 99 & 4\ 81 & 5\ 64 & 6\ 66 & 8\ 80 & 2\ 80 & 7\ 69 & 9\ 62 & 3\ 79 & 0\ 88 \\
7\ 50 & 1\ 86 & 4\ 97 & 3\ 96 & 0\ 95 & 8\ 97 & 2\ 66 & 5\ 99 & 6\ 52 & 9\ 71 \\
4\ 98 & 6\ 73 & 3\ 82 & 2\ 51 & 1\ 71 & 5\ 94 & 7\ 85 & 0\ 62 & 8\ 95 & 9\ 79 \\
0\ 94 & 6\ 71 & 3\ 81 & 7\ 85 & 1\ 66 & 2\ 90 & 4\ 76 & 5\ 58 & 8\ 93 & 9\ 97 \\
3\ 50 & 0\ 59 & 1\ 82 & 8\ 67 & 7\ 56 & 9\ 96 & 6\ 58 & 4\ 81 & 5\ 59 & 2\ 96
\end{bmatrix}$$

The identification of the elements from the previous input data example is shown in the following matrix.

	m_0	t_0	m_1	t_1	m_2	t_2	m_3	t_3	m_4	t_4	m_5	t_5	m_6	t_6	m_7	t_7	m_8	t_8	m_9	t_9
j_0	4	88	8	68	6	94	5	99	1	67	2	89	9	77	7	99	0	86	3	92
j_1	5	72	3	50	6	69	4	75	2	94	8	66	0	92	1	82	7	94	9	63
j_2	9	83	8	61	0	83	1	65	6	64	5	85	7	78	4	85	2	55	3	77
j_3	7	94	2	68	1	61	4	99	3	54	6	75	5	66	0	76	9	63	8	67
j_4	3	69	4	88	9	82	8	95	0	99	2	67	6	95	5	68	7	67	1	86
j_5	1	99	4	81	5	64	6	66	8	80	2	80	7	69	9	62	3	79	0	88
j_6	7	50	1	86	4	97	3	96	0	95	8	97	2	66	5	99	6	52	9	71
j_7	4	98	6	73	3	82	2	51	1	71	5	94	7	85	0	62	8	95	9	79
j_8	0	94	6	71	3	81	7	85	1	66	2	90	4	76	5	58	8	93	9	97
j_9	3	50	0	59	1	82	8	67	7	56	9	96	6	58	4	81	5	59	2	96

By parsing this elements the result is a list of lists where each list represent one job (a row of the matrix) and each job contains a number of tuples with the data (m_i, t_i).

$$\begin{bmatrix}
[(4,88) & (8,68) & (6,94) & (5,99) & (1,67) & (2,89) & (9,77) & (7,99) & (0,86) & (3,92)] \\
[(5,72) & (3,50) & (6,69) & (4,75) & (2,94) & (8,66) & (0,92) & (1,82) & (7,94) & (9,63)] \\
[(9,83) & (8,61) & (0,83) & (1,65) & (6,64) & (5,85) & (7,78) & (4,85) & (2,55) & (3,77)] \\
[(7,94) & (2,68) & (1,61) & (4,99) & (3,54) & (6,75) & (5,66) & (0,76) & (9,63) & (8,67)] \\
[(3,69) & (4,88) & (9,82) & (8,95) & (0,99) & (2,67) & (6,95) & (5,68) & (7,67) & (1,86)] \\
[(1,99) & (4,81) & (5,64) & (6,66) & (8,80) & (2,80) & (7,69) & (9,62) & (3,79) & (0,88)] \\
[(7,50) & (1,86) & (4,97) & (3,96) & (0,95) & (8,97) & (2,66) & (5,99) & (6,52) & (9,71)] \\
[(4,98) & (6,73) & (3,82) & (2,51) & (1,71) & (5,94) & (7,85) & (0,62) & (8,95) & (9,79)] \\
[(0,94) & (6,71) & (3,81) & (7,85) & (1,66) & (2,90) & (4,76) & (5,58) & (8,93) & (9,97)] \\
[(3,50) & (0,59) & (1,82) & (8,67) & (7,56) & (9,96) & (6,58) & (4,81) & (5,59) & (2,96)]
\end{bmatrix}$$

3.3 Genetic algorithm

The genetic algorithm follows the next steps:

1. Initialize Population: The population P is initialized.
2. Evaluate Initial Fitness: The fitness F of all individuals in the population is evaluated.
3. Start Generations Loop: A loop begins to run for the specified number of generations, where each iteration represents one generation.

4. Create New Population: A new population P' is created for the current generation.
5. Select Parents: For each pair of individuals, it is selected two parents based on the chosen selection method (roulette-wheel or tournament).
6. Crossover: The selected parents undergo crossover to create two offspring (new solutions).
7. Mutation: The offspring undergo mutation to generate new possible solutions and population diversity.
8. Add to New Population: The mutated offspring are added to the new population P'.
9. Apply Elitism: Elitism is applied to ensure the best individuals from the previous generation are carried over.
10. Update Population: The current population P is updated to the new population P'.
11. Evaluate Fitness: The fitness of the new population P is evaluated again.
12. End of Generation Loop: The process repeats for the specified number of generations.
13. Store Results: After all generations, the final population P and its fitness F are stored.

While this code only has the main structure the implementations are in the rest of internal methods of the class. This is done to keep the modularity, improve the debug and make easier to modify in the future.

```python
def run_genetic_algorithm(self, num_generations, population_size, selection_method='roulette-wheel', crossover_method='one-point', mutation_method='flip-one', mutation_probability=0.5, elitism_number=1):
    # Initialize population P
    _P = self._initialize_population(population_size)

    # Evaluate fitness of all individuals in P
    _F = self._evaluate_fitness(_P)

    # Iterate over generations
    for _ in range(num_generations):
        _P_prime = []
        # Iterate over pairs of individuals
        for _ in range(population_size//2):
            # Selection
            _c_a, _c_b = self._selection(_P, _F, selection_method)
            # Crossover
            _c_prime_a, _c_prime_b = self._crossover(_c_a, _c_b, crossover_method)
            # Mutation
            _c_prime_mutated_a = self._mutate(_c_prime_a, mutation_method, mutation_probability)
            _c_prime_mutated_b = self._mutate(_c_prime_b, mutation_method, mutation_probability)
            # Add to the new population
            _P_prime.append(_c_prime_mutated_a)
            _P_prime.append(_c_prime_mutated_b)
        # Elitism
        _P_prime = self._elitism(_P, _F, _P_prime, elitism_number)
        # Update population
        _P = _P_prime
        # Evaluate fitness of all individuals in P
        _F = self._evaluate_fitness(_P)
    self.solutions = _P
    self.solutions_fitness = _F
    self.best_solution, self.best_solution_fitness = self._get_best_solution(_P, _F)
```

Figure 1: Python code of the main genetic algorithm implementation

3.4 Convert to chromosomes

As mentioned before a chromosome is represented by a sequence of 0s and 1s such $c \in \{0,1\}^{n_{machines}}$. To get this result the format used has been (job_index, operation_index), the previous list is flattened (5).

$$(i,j) \mid i \in \{0,1,\ldots,n_{jobs}-1\},\ j \in \{0,1,\ldots,n_{machines}-1\} \tag{5}$$

The previous example looks as it follows after the transformation.

$$\{(0,0),(0,1),\ldots,(0,9),$$
$$(1,0),(1,1),\ldots,(9,9)\}$$

This transformation is done by iterating over all the elements and saving their indexes in a tuple.

```python
# GENETIC ALGORITHM
def _initialize_population(self, population_size):
    # Output variable
    _population = []

    # Flatten the jobs to operation identifiers int he format (job_index, operation_index)
    _operation_ids = []
    for _job_idx, _job in enumerate(self.jobs):
        for _operation_idx in range(len(_job)):
            _operation_ids.append((_job_idx, _operation_idx))
```

Figure 2: Python code to convert the input data into chromosomes

3.5 Creating the initial population

To create the intial population it is necessary to transform the initial data into a chromosomes. The initial population is created randomly by creating permutations. Basically, it is generated random permutations of operations while respecting job operation dependencies, ensuring that each operation is scheduled only after its preceding operation for the same job. This will always generate valid permutations.

The previous explained chromosome conversion and the creation of the initial population are done in the same method _initialize_population.

```python
# GENETIC ALGORITHM
def _initialize_population(self, population_size):
    # Output variable
    _population = []

    # Flatten the jobs to operation identifiers int he format (job_index, operation_index)
    _operation_ids = []
    for _job_idx, _job in enumerate(self.jobs):
        for _operation_idx in range(len(_job)):
            _operation_ids.append((_job_idx, _operation_idx))

    # Generate random permutations taking in count restrictions
    for _ in range(population_size):
        _not_processed_operations = _operation_ids[:]
        _current_permutation = []
        while len(_not_processed_operations) > 0:
            _valid_operations = []
            for _operation in _not_processed_operations:
                _job_idx, _operation_idx = _operation
                if _operation_idx == 0 or ((_job_idx, _operation_idx - 1) in _current_permutation):
                    _valid_operations.append(_operation)
            _selected_operation = random.choice(_valid_operations)
            _current_permutation.append(_selected_operation)
            _not_processed_operations.remove(_selected_operation)
        _population.append(_current_permutation)
    return _population
```

Figure 3: Python code that genereates the initial populations

3.6 Evaluate fitness

Evaluating the fitness of the population is important to determine how well each solution satisfies the problem's constraints and to guide the selection of the best solutions for optimization. In this case, the fitness of each solution is evaluated by calculating the makespan, which is the total time required to complete all jobs. The evaluation ensures that each operation respects two key constraints: a machine cannot start a new operation until it is free, and a job cannot proceed to its next operation until the previous one is completed. These constraints are handled by calculating the earliest start time for each operation based on the availability of the machine and the job's completion. The fitness value is the maximum end time across all machines, representing the makespan.

To ensure that operations are scheduled according to the constraints, the function calculates the earliest possible start time for the operation. This is determined by taking the maximum of two values: the time when the machine becomes free (_machine_end_times[_machine]) and the time when the job becomes available (_job_end_times[_job_idx]). The operation starts at the later of these two times to ensure both the machine and the job are ready.

Once the start time is determined, the end time of the operation is calculated by adding the processing time. The machine and job availability are then updated based on the operation's end time. This step ensures that the machine and job schedules reflect the completion of this operation.

After all operations for the individual have been scheduled, the makespan is calculated as the maximum end time across all machines. This makespan represents the total time required to finish all operations in the schedule. The fitness of the individual is the value of this makespan, with a lower makespan indicating a better solution.

```python
def _evaluate_fitness(self, population):
    # Output variable
    _fitness = []

    # Calculate for each different solution the makespan
    for _individual in population:
        # Initialize the times for each sequence
        _machine_end_times = [0] * self.number_of_machines
        _job_end_times = [0] * self.number_of_jobs
        for _operation in _individual:
            _job_idx, _operation_idx = _operation
            _machine, _processing_time = self.jobs[_job_idx][_operation_idx]
            # The machine cannot start a new job if the machine is not free or the previous job is not finished
            # To satisfy both constraints the largest time is selected
            _earliest_start = max(_machine_end_times[_machine], _job_end_times[_job_idx])
            # The operation starts at the _earliest_start
            _start_time = _earliest_start
            # The operation ends after the processing time
            _end_time = _start_time + _processing_time
            # Update machine availability
            _machine_end_times[_machine] = _end_time
            # Update job availability
            _job_end_times[_job_idx] = _end_time
        # The maximum end time across all machines to finish defines the total processing time
        _fitness.append(max(_machine_end_times))
    return _fitness
```

Figure 4: Python code that calculates the fitness of the population

3.7 Selection

Selection ensures that better solutions have a higher chance of being passed on to the next generation. It drives the evolution towards optimal solutions by prioritizing individuals with higher fitness. The selection function chooses two individuals from the population based on their fitness, using either the tournament or roulette wheel method. It ensures that the most suitable candidates are selected for the next generation.

In the roulette wheel method, individuals are selected based on their fitness proportions. The higher an individual's fitness, the more likely it is to be selected. The selection is done by calculating the fitness percentage of each individual, followed by a random choice based on these probabilities.

In the tournament method, a subset of individuals is randomly selected from the population, and the fittest individual within this subset is chosen. This process is repeated to select two individuals, ensuring that the fittest candidates within the subset have a higher chance of selection.

```python
def _selection(self, population, fitness, method='roulette-wheel'):
    if method == 'tournament':
        _c = []
        for _ in range(2):
            # Select a random k individuals
            _k = random.randrange(len(population)) + 1
            # Create a list with the possible indexes for the selection
            _possible_indexes = list(range(len(fitness)))
            # Select _k indexes
            _selected_index = random.choices(_possible_indexes, k=_k)
            # Extract elements at specific indexes
            _selected_fitness = [fitness[i] for i in _selected_index]
            # Select the fittest one
            _max_fitness_value = max(_selected_fitness)
            _max_fitness_index = _selected_fitness.index(_max_fitness_value)
            # Selected chromosomes
            _c.append(population[_selected_index[_max_fitness_index]])
        return (_c[0], _c[1])
    else:
        # Roulette wheel selection method
        # Initialize the pair of chromosomes selected indexes
        _index_selected_a = 0
        _index_selected_b = 0
        # Calculate the total fitness to obtain the percentages
        _total_fitness = sum(fitness)
        # Calculate the percentages of each chromosome fitness
        _fitness_percentages = [_value / _total_fitness for _value in fitness]
        # Create a list with the possible indexes for the selection
        _possible_indexes = list(range(len(fitness)))
        # Select the index a using the probabilites
        _index_selected_a = random.choices(_possible_indexes, _fitness_percentages, k=1)[0]
        # Delete the selected choice from the list of possible indexes
        _possible_indexes.pop(_index_selected_a)
        _fitness_percentages.pop(_index_selected_a)
        # Select the index b from the remaining possible indexes using the probabilities
        _index_selected_b = random.choices(_possible_indexes, _fitness_percentages, k=1)[0]
        # Return the two corresponding chromosomes to the selected indexes a and b
        return (population[_index_selected_a], population[_index_selected_b])
```

Figure 5: Python code that implements the selection methods

3.8 Crossover

Crossovercombines the genetic material of two parents to produce offspring with characteristics from both. This helps explore new solutions and maintain genetic diversity in the population, driving the algorithm towards optimal results. The crossover function generates offspring by exchanging genetic material from two parent individuals. It allows different crossover methods, such as one-point and uniform crossover, to combine the parent solutions and create diverse offspring for the next generation.

In the uniform crossover method, each operation in the chromosomes is considered for swapping. With a 50% probability, an operation from one parent is exchanged with the corresponding operation from the other parent. This method ensures that parts from both parents are mixed in the offspring.

In the one-point crossover method, a random crossover point is selected. The portion of the chromosome after this point is swapped between the two parents, generating two offspring that inherit parts of each parent before and after the selected point.

```python
def _crossover(self, individual_1, individual_2, method='one-point'):
    if method == 'uniform':
        # Intialize the pair of chromosomes for the crossover
        _c_a = individual_1.copy()
        _c_b = individual_2.copy()
        # Iterate over all the operations in the pair of chromosomes
        for _operation_idx, _ in enumerate(_c_a):
            # Swap the operations with a probability of 50%
            if random.random() < 0.5:
                _operation_1 = _c_a[_operation_idx]
                _operation_2 = _c_b[_operation_idx]
                _c_a[_operation_idx] = _operation_2
                _c_b[_operation_idx] = _operation_1
        return (_c_a, _c_b)
    else:
        # One-point crossover method
        # Intialize the pair of chromosomes for the crossover
        _c_a = individual_1.copy()
        _c_b = individual_2.copy()
        # Select a random position between 1 and the last postion of the chromosome
        _random_position = random.randint(1, len(_c_a) - 1)
        # Swap the parts between the chromosomes
        _c_a[_random_position:], _c_b[_random_position:] = individual_2[_random_position:], individual_1[_random_position:]
        # Rteurn the corresponding chromosomes
        return (_c_a, _c_b)
```

Figure 6: Python code that implements the crossover methods

3.9 Mutation

Mutation introduces random changes to individuals, helping to explore new solutions outside the current population's genetic pool. This helps prevent premature convergence and maintains diversity, ensuring the algorithm doesn't get stuck in local optima. The mutation function applies random changes to an individual's chromosome. It provides different mutation methods, such as "flip-probabilistic" and "flip-one", to introduce variety by flipping parts of the chromosome and altering the solution.

In the flip-probabilistic mutation method, each operation in the chromosome has a certain probability of being flipped. If the random number is less than the probability, the operation's bits are negated, introducing variability and increasing genetic diversity.

In the flip-one mutation method, a random position in the chromosome is selected, and its value is flipped. This introduces a single mutation in the chromosome, altering the solution with minimal disruption while maintaining overall structure.

```python
def _mutate(self, individual, method, probability=0.5):
    if method == 'flip-probabilistic':
        # Intialize the chromosome for the mutation
        _c = individual.copy()
        # Iterate over all the operations in the pair of chromosomes
        for _operation_idx, _ in enumerate(_c):
            # Swap the operations with a probability of 50%
            if random.random() < probability:
                # Flip the bits int he chromosome
                _negated_tuple = tuple(1 - x for x in _c[_operation_idx])
                # Insert the tuple again
                _c[_operation_idx] = _negated_tuple
        # Return the mutated chromosome
        return _c
    else:
        # Flip one mutation method
        # Intialize the chromosome for the mutation
        _c = individual.copy()
        # Select a random position between 0 and the last postion of the chromosome
        _random_position = random.randint(0, len(_c) - 1)
        # Flip the bits int he chromosome
        _negated_tuple = tuple(1 - x for x in _c[_random_position])
        # Insert the tuple again
        _c[_random_position] = _negated_tuple
        # Return the mutated chromosome
        return _c
```

Figure 7: Python code that implements the crossover methods

3.10 Elitism

Elitism ensures the best solutions are carried forward to the next generation. Without elitism, there's a risk of losing the best solution due to random crossover or mutation, thus preventing progress. This process helps to retain high-quality individuals and speeds up convergence. This implementation aims to preserve the best-performing individuals from one generation to the next. The elite individuals are selected from the current population based on their fitness, ensuring that they survive into the next generation. This increases the likelihood of finding better solutions over time.

The function _elitism takes two populations and their fitness values. It finds the number of individuals with the smallest fitness values (best solutions) from population_1. These best individuals are added to population_2, which is returned as the new population. The method uses heapq.nsmallest() to efficiently select the best individuals based on their fitness values.

```
def _elitism(self, popuplation_1, population_1_fitness, population_2, number=1):
    _new_P_prime = population_2.copy()
    _n_largest = heapq.nsmallest(number, enumerate(population_1_fitness), key=lambda x: x[1])
    _smallest_indexes = [_index for _index, _ in _n_largest]
    for _index in _smallest_indexes:
            _new_P_prime.append(popuplation_1[_index])
    return _new_P_prime
```

Figure 8: Python code that implements the elitism method

3.11 Validate solution

Validating ensures the generated solutions respect the problem's constraints. For the Job-Shop Scheduling Problem, a valid solution must follow job precedence and machine availability rules. If solutions violate these constraints, they become irrelevant, reducing the chances of finding a true optimal solution. By filtering out invalid solutions, the algorithm focuses on feasible solutions. After some mutation, crossover and selection steps some solutions could become invalid and those cannot be taken in count for the final selection of the fitest one. The _validate_solution function checks whether a given solution satisfies the constraints of the Job-Shop Scheduling Problem. It verifies the job precedence and ensures machines are available before processing operations. If any constraint is violated, it returns False. The _get_best_solution function refines the population by removing invalid solutions and selects the best valid solution based on fitness. The filtered population is then used to track and update the best solution found.

The _validate_solution method iterates over each operation in a solution. For each operation, it checks if job precedence is maintained and if machines are available to process the operation without conflicts. If any of the constraints are violated, the solution is invalid, and the method returns False. Otherwise, it updates machine and job end times and returns True.

The _get_best_solution function first filters out invalid solutions from the population using the _validate_solution method. It then identifies the best solution among the valid ones by selecting the one with the lowest fitness value. If no valid solutions remain, it returns a placeholder value [-1, -1] to indicate failure. The function ensures that only feasible solutions are considered.

```python
def _validate_solution(self, individual):
    # Calculate for each different solution the makespan
    _machine_end_times = [0] * self.number_of_machines
    _job_end_times = [0] * self.number_of_jobs
    for _operation in individual:
        _job_idx, _operation_idx = _operation
        _machine, _processing_time = self.jobs[_job_idx][_operation_idx]
        # Ensure operations within a job are in correct order
        if _operation_idx > 0 and (_job_idx, _operation_idx - 1) not in individual:
            return False  # Job precedence violated
        _earliest_start = max(_machine_end_times[_machine], _job_end_times[_job_idx])
        # The operation starts at the _earliest_start
        _start_time = _earliest_start
        # The operation ends after the processing time
        _end_time = _start_time + _processing_time
        # Ensure machine is available at the start time and check for overlap
        if _machine_end_times[_machine] > _start_time:
            return False  # Machine availability violated
        # Update machine availability
        _machine_end_times[_machine] = _end_time
        # Update job availability
        _job_end_times[_job_idx] = _end_time
    return True

def _get_best_solution(self, population, fitness):
    _filtered_P = population.copy()
    _filtered_F = fitness.copy()
    for _idx in range(len(_filtered_P)-1, -1, -1):  # Iterate in reverse order
        if not self._validate_solution(_filtered_P[_idx]):
            _filtered_P.pop(_idx)
            _filtered_F.pop(_idx)

    if len(_filtered_P) > 0 and len(_filtered_F) > 0:
        _best_solution = []
        _best_solution_fitness = min(_filtered_F)
        _best_solution_index = _filtered_F.index(_best_solution_fitness)
        _best_solution.append(_filtered_P[_best_solution_index])
        self.validated_solutions = _filtered_P
        self.validated_solutions_fitness = _filtered_F
        return(_best_solution, _best_solution_fitness)
    else:
        # No solution found
        return([-1, -1], -1)
```

Figure 9: Python code that implements the validation methods

4 Results

Some validations have been done using datasets from https://people.brunel.ac.uk/ mastjjb/jeb/orlib/files/jobshop1.txt.

4.1 Instance ft06 - Fisher and Thompson 6x6 instance

This instance has 6 machines and 6 jobs. Tg

Number of Generations	Initial Population Size	Selection Method	Crossover Method	Mutation Method	Fitness
20	100	Roulette wheel	One-point	Bit-Flip	56
20	100	Tournament	One-point	Bit-Flip	68
20	100	Roulette wheel	Uniform	Bit-Flip	60
20	100	Tournament	Uniform	Bit-Flip	69
20	100	Roulette wheel	One-point	Probabilistic (0.1)	68
20	100	Tournament	One-point	Probabilistic (0.1)	69

Table 1: Results for instance ft06 with different parameter combinations

The last test (tournament, one-point and probabilistic (0.1)) has been represented in a gnatt chart for better understanding of the solutions.

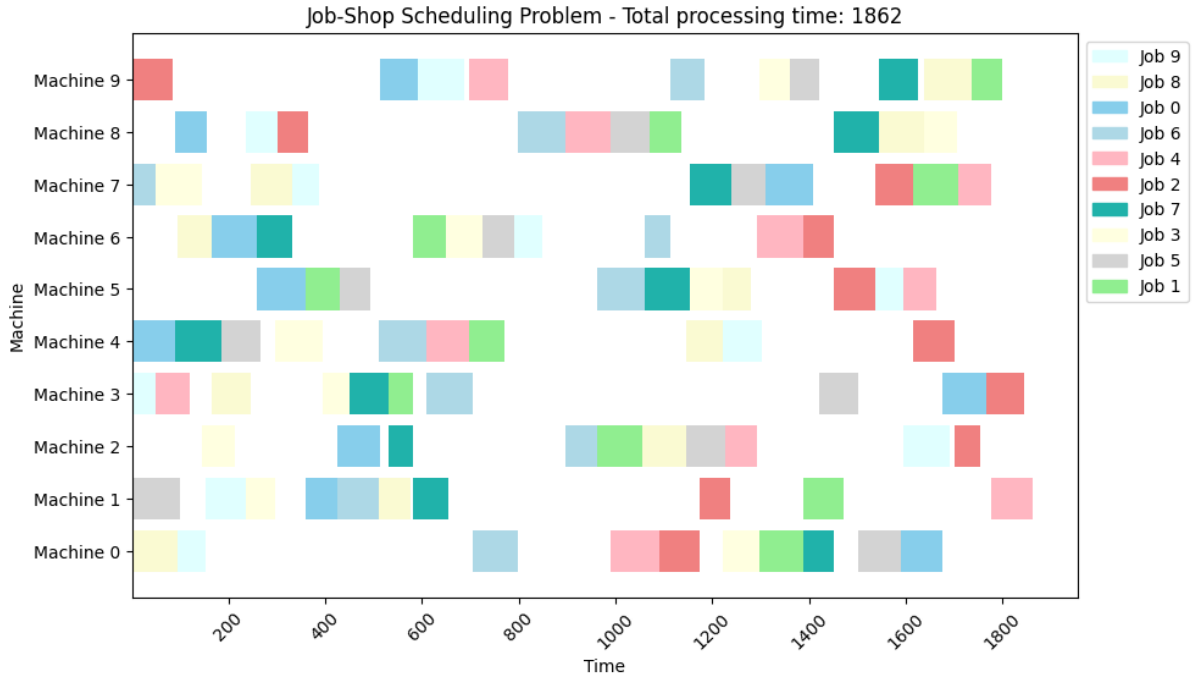

Figure 10: Gnatt chart of the last entry in the table **??**

4.2 Instance abz5 - Adams, Balas, and Zawack 10x10 instance

This instance has 10 machines and 10 jobs.

Number of Generations	Initial Population Size	Selection Method	Crossover Method	Mutation Method	Fitness
20	100	Roulette wheel	One-point	Bit-Flip	1495
20	100	Tournament	One-point	Bit-Flip	1796
20	100	Roulette wheel	Uniform	Bit-Flip	1699
20	100	Tournament	Uniform	Bit-Flip	1773
20	100	Roulette wheel	One-point	Probabilistic (0.1)	1913
20	100	Tournament	One-point	Probabilistic (0.1)	1862

Table 2: Results for instance la36 with different parameters

The last test (tournament, one-point and probabilistic (0.1)) has been represented in a gnatt chart for better understanding of the solutions.

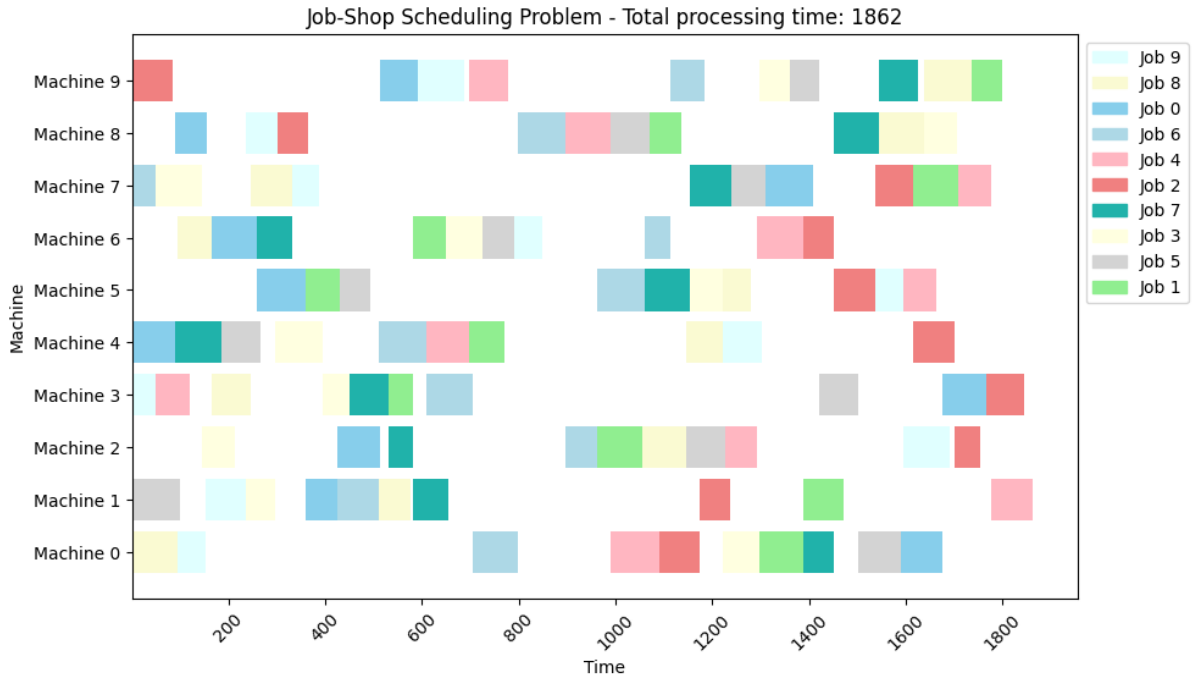

Figure 11: Gnatt chart of the last entry in the table **??**

4.3 Instance la36 - Lawrence 15x15 instance

This instance has 15 machines and 15 jobs.

Number of Generations	Initial Population Size	Selection Method	Crossover Method	Mutation Method	Fitness
20	100	Roulette wheel	One-point	Bit-Flip	2042
20	100	Tournament	One-point	Bit-Flip	2420
20	100	Roulette wheel	Uniform	Bit-Flip	2387
20	100	Tournament	Uniform	Bit-Flip	2287
20	100	Roulette wheel	One-point	Probabilistic (0.1)	2227
20	100	Tournament	One-point	Probabilistic (0.1)	2281

Table 3: Results for instance la36 with different parameters

The last test (tournament, one-point and probabilistic (0.1)) has been represented in a gnatt chart for better understanding of the solutions.

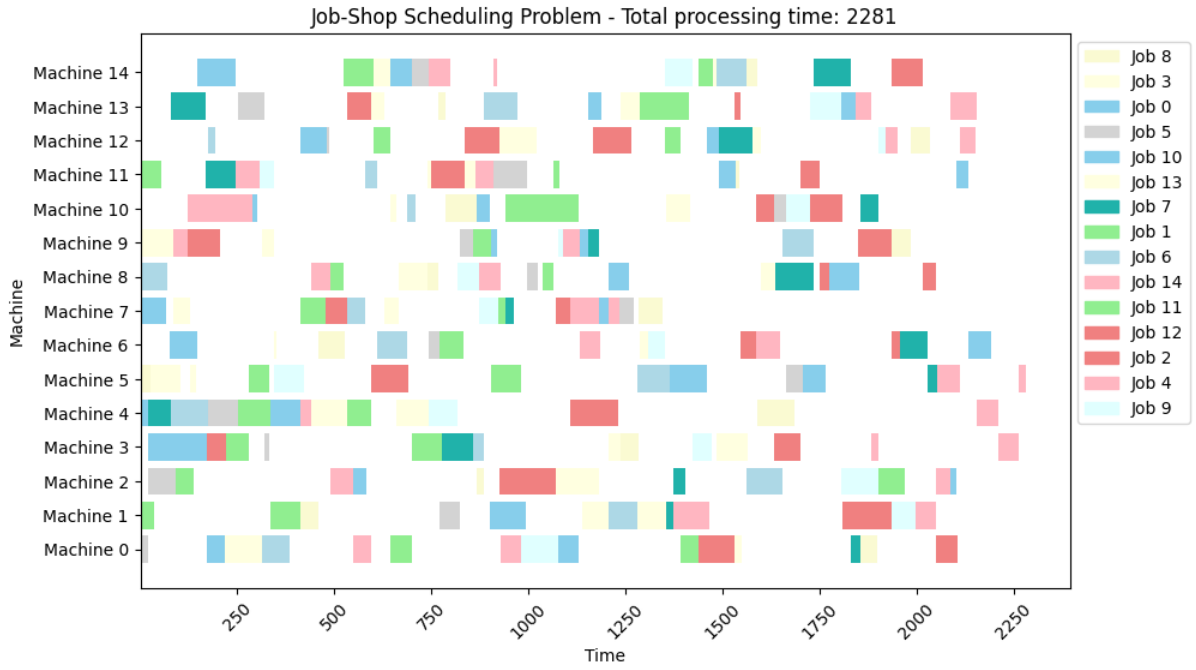

Figure 12: Gnatt chart of the last entry in the table **??**

4.4 Minimum travel tiempo across generations

After carrying several tests, it became apparent that many of the solutions generated across multiple generations were either identical or extremely similar to the initial solutions. This phenomenon primarily stems from the fact that the crossover and mutation operations frequently produced invalid solutions, which were subsequently discarded during the validation phase. The high number of invalid solutions can be attributed to the nature of these genetic operations, which often do not respect the problem constraints, such as job precedence or machine availability, thus leading to infeasible solutions.

The persistent generation of invalid solutions indicates a need for more effective genetic operators—specifically, crossover and mutation methods better tailored to the constraints and characteristics of the Job Shop Scheduling Problem (JSSP). A key challenge here is ensuring that the crossover and mutation processes generate solutions that remain valid and feasible throughout the evolutionary process.

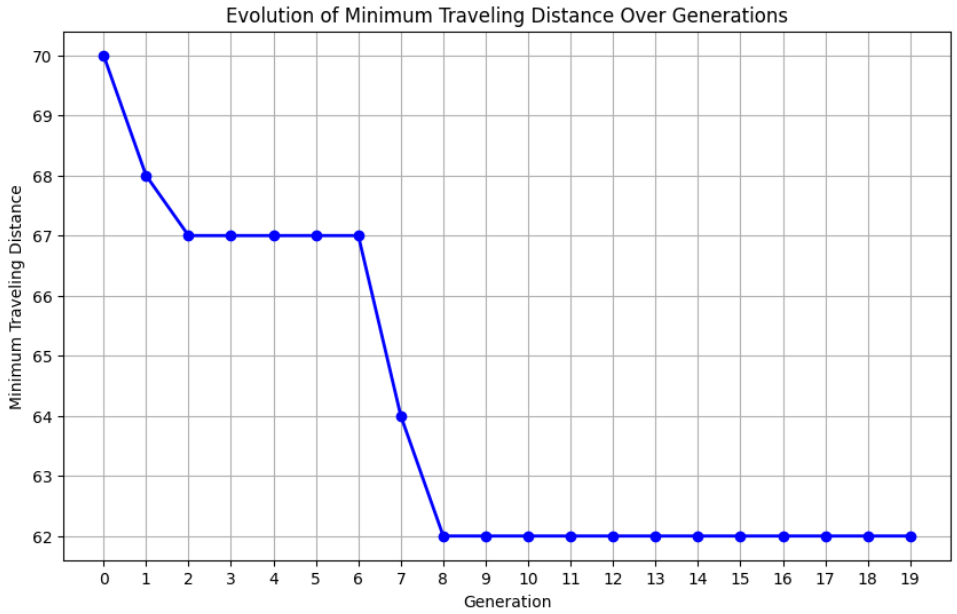

Figure 13: Minimum travel distance for a ft06 instance execution

To address this issue, selecting more appropriate crossover and mutation methods should be explored. For example, order-based crossover methods, like OX (Order Crossover) or PMX (Partially Matched Crossover), have been shown to perform well in combinatorial problems, especially those involving ordered sets like job schedules. These methods are designed to preserve the relative order of genes in the offspring, which helps maintain the feasibility of the solution. It would be a possibility to explore this ways.

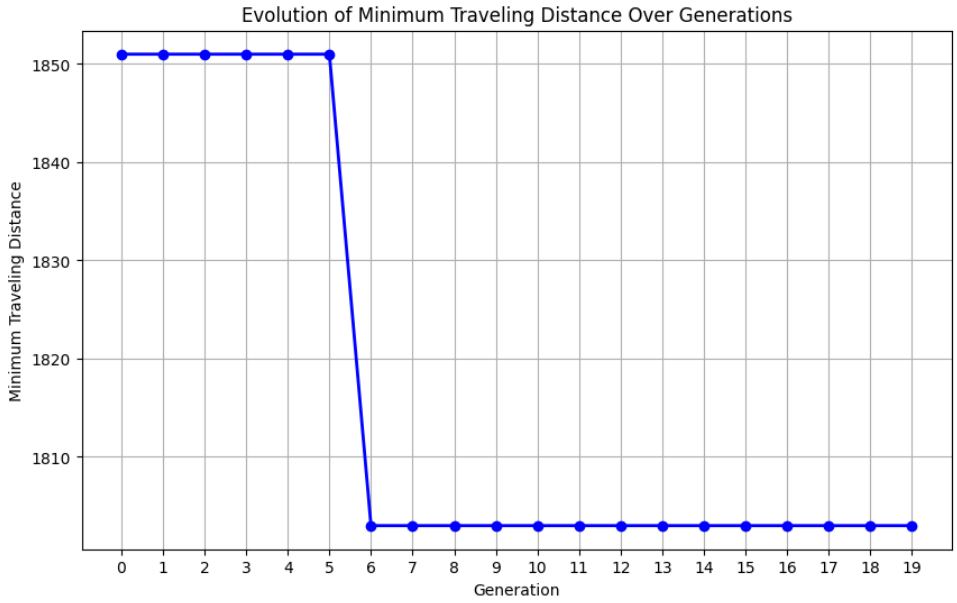

Figure 14: Minimum travel distance for a abz5 instance execution

In terms of mutation, swap mutation is another useful technique for job-shop problems, where

two jobs are swapped within a machine or between machines. This type of mutation e nsures that the job schedule remains valid by keeping the sequence intact while introducing variability. Alternatively, insertion-based mutation, where a job is moved to a different position in the schedule, can help explore the search space more efficiently.

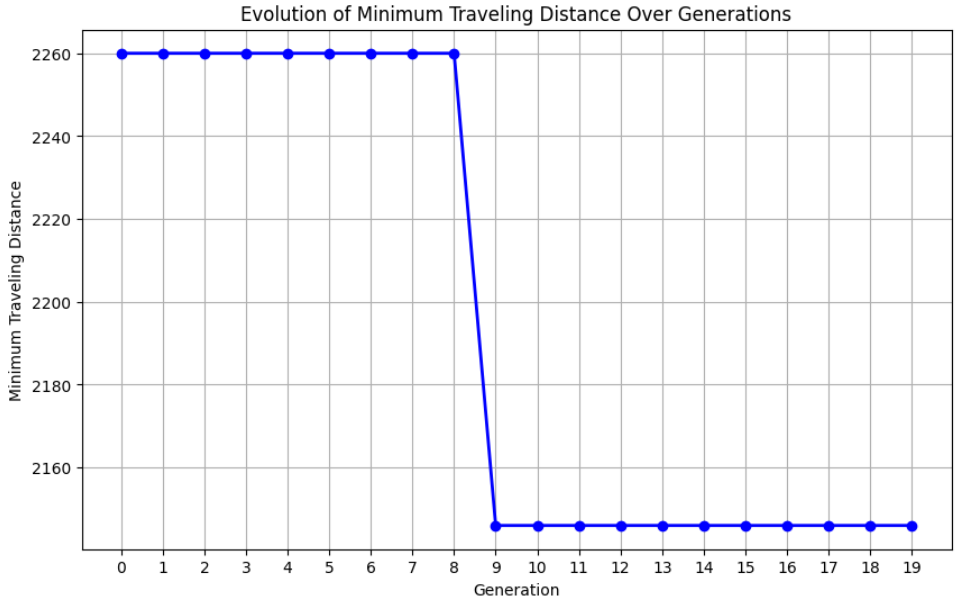

Figure 15: Minimum travel distance for a la36 instance execution

In conclusion, the frequent generation of invalid solutions in the current approach emphasizes the importance of selecting crossover and mutation methods that are specifically designed to maintain the feasibility of the solution. By incorporating these more sophisticated genetic operators, probably we will get better results.

5 Conclusion

Genetic algorithms (GA) show potential for solving the Job Shop Scheduling Problem. They can explore large search spaces and find good solutions quickly. However, the crossover and mutation methods must be carefully chosen to produce valid solutions. Adjusting the population size and fine-tuning operators can improve performance. Despite challenges like premature convergence, experimenting with different GA variations may lead to better results.

5.1 Further steps

The genetic algorithm (GA) showed that many solutions in later generations were similar or the same as the initial solutions. This happened because the crossover and mutation methods often produced invalid solutions. These solutions were discarded during validation. Validation is crucial in filtering out invalid solutions. However, this process shows the need for better solution generation methods. The current crossover and mutation strategies often require discarding too many solutions. In other words, the crossover and mutation methods used need to be more suited to the problem. They should generate more valid solutions. Due to time restriction I could not explore this posibilities.

References

[1] Wikipedia contributors, *Genetic algorithm*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Genetic_algorithm, accessed January 18, 2025.

[2] Wikipedia contributors, *Evolutionary computation*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Evolutionary_computation, accessed January 18, 2025.

[3] Wikipedia contributors, *Job-shop scheduling*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Job-shop_scheduling, accessed January 19, 2025.

[4] Wikipedia contributors, *Selection (evolutionary algorithm)*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Selection_(evolutionary_algorithm), accessed January 19, 2025.

[5] Wikipedia contributors, *Crossover (evolutionary algorithm)*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Crossover_(evolutionary_algorithm), accessed January 19, 2025.

[6] Wikipedia contributors, *Mutation (evolutionary algorithm)*, Wikipedia, The Free Encyclopedia, Available at https://en.wikipedia.org/wiki/Mutation_(evolutionary_algorithm), accessed January 19, 2025.