ACTIVITY 2

Optimization with Genetic Algorithms
Job Shop Scheduling Problem

(by Victoria Joven)

---

https://github.com/victoriajoven/NeuronalActivity_2

Index

1. Chromosome and algorithmic adaptations

This section explains Genetic Algorithm (GA) implemented in this project adapts the classical formulation taught in the course to the specific requirements of the Job Shop Scheduling Problem (JSSP). The content follows the theoretical foundations introduced in the lectures—population-based search, selection pressure, recombination, mutation, elitism, and convergence—while explicitly linking each concept to the corresponding Python implementation.

1.1 Chromosome

Definition

In the course (Neuronal and Evolutionary Computation), chromosomes are defined as vectors of genes, typically binary strings $c \in \{0,1\}^m$. Evolutionary algorithms are not limited to binary chromosomes and can be different parameters as integer sequences, trees, or similar structures. This change is necessary when a problem requires other representations that binary encodings cannot represent.

In JSSP each job consists of a fixed sequence of operations, and the chromosome must encode an ordering of operations that respects job precedence while allowing the GA has to use relationships between jobs.

The implementation use a **job-based permutation encoding**, where the **chromosome is a sequence of job identifiers**. If job j has k operations, it appears k times in the chromosome. The position of each occurrence determines the order in which its operations are scheduled.

Code implementation

The chromosome and population representation is implemented in:

- `src/ga/chromosome.py`
- `src/ga/population.py → Population.initialize()`

The initialization procedure constructs the base sequence of job identifiers and shuffles it to generate the initial population, following the “random initialization”.

1.2 Fitness evaluation

Definition

Fitness is defined as a non-negative value that quantifies the quality of an individual. Fitness can also be defined as a numerical score that measures how good a solution (individual) is at solving a problem (as in the evolution of species, survival, and reproduction).

GA can solve minimization problems by transforming the fitness or adapting the selection operator.

Code implementation

The fitness function is the **makespan**, computed by simulating the schedule.

The simulation is implemented in:

- `src/evaluation/makespan.py → compute_makespan()`

1.3 Selection techniques

In the NEC course, we have seen four selection schemes: roulette wheel, rank selection, tournament selection, and stochastic universal sampling. Two of these are implemented in this project (tournament and roulette wheel).

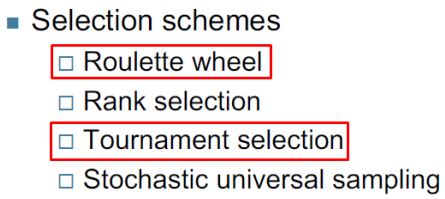


Figure 1. Selection techniques used on experiments

1.3.1 Tournament selection

Definition

Tournament selection:

- Selects K individuals at random,
- Chooses the best among them,
- Maintains diversity,
- Is robust to fitness scaling,
- Avoids the sensitivity issues (random of roulette wheel selection).

Code implementation

Implemented in:

- `src/ga/techniques/selection.py → TournamentSelection`

The method samples a subset of individuals and selects the **one with the lowest makespan**, consistent with the minimization objective.

1.3.2 Roulette wheel selection

Definition

Roulette wheel selection assigns a probability proportional to fitness:

- Its sensitivity to fitness distribution,
- The requirement of non-negative fitness values,
- Its tendency to reduce diversity when fitness values differ greatly.

Code implementation

Implemented in:

- `src/ga/techniques/selection.py → RouletteWheelSelection`

```
# Ensure even number of parents
if pop_size % 2 != 0:
    pop_size -= 1

# Fitness values (minimization problem)
fitnesses = np.array([c.fitness for c in pop])

# Lower fitness = higher probability
max_fitness = fitnesses.max()
adjusted_fitness = max_fitness - fitnesses + 1e-6

probabilities = adjusted_fitness / adjusted_fitness.sum()

for _ in range(pop_size):
    selected = np.random.choice(pop, p=probabilities)
    parents.append(selected)
```

The code adapts the method for minimization by inverting fitness values. The comments indicate the algorithm applied. I have also added validation preconditions to minimize errors.

1.4 Crossover techniques

In unit 6, we looked at the concept of the crossover with techniques one-point, two-point and uniform for binary encodings, but also states that for non-binary or permutation encodings, crossover must be adapted.

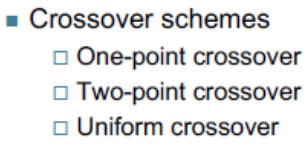


Figure 2. Crossover schemes seen in NEC classes

While researching, I have seen crossover algorithms that I will explain below, already implemented in Python with very optimal results.

1.4.1 Precedence preserving crossover (PPX)

Definition

For permutation-based problems, crossover complies with:

- Mimics reproduction (genetic recombination),
- Create two offspring from two parents
- Used to create next generation population

Code implementation

Implemented in:

- `src/ga/techniques/crossover.py → PrecedencePreservingCrossover`

```python
def apply(self, parents):
    offspring = []

    for i in range(0, len(parents) - 1, 2):
        p1 = parents[i].genes
        p2 = parents[i + 1].genes

        child = []
        r1 = p1.copy()
        r2 = p2.copy()

        while r1:
            if random.random() < 0.5:
                gene = r1.pop(0)
                r2.remove(gene)
            else:
                gene = r2.pop(0)
                r1.remove(gene)
            child.append(gene)

        offspring.append(Chromosome(child))

    return offspring
```

The PPX technique **selects genes alternately from the parents while removing them from both lists**, ensuring feasibility and previous premises of crossover.

1.4.2 Job-based crossover (JBX)

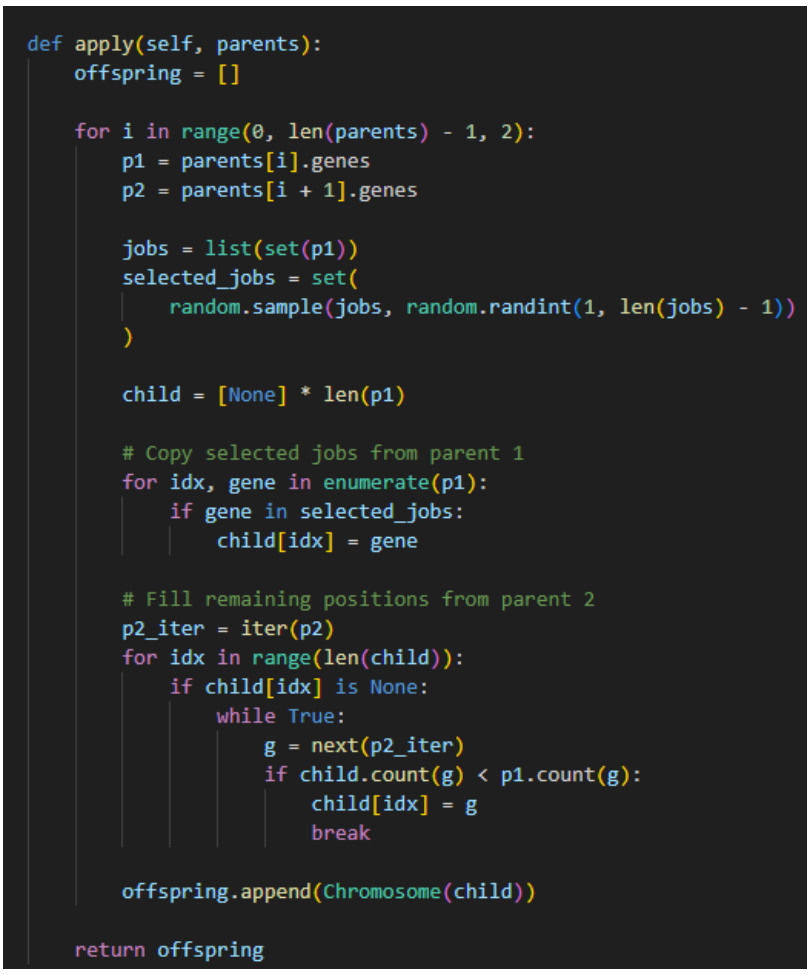
Definition

This tecnique preserves positional information for a subset of jobs and fills remaining positions from the other parent. It follows the course's principle that crossover must recombine genetic material without violating the constraints of the encoding.

Code implementation

Implemented in:

- `src/ga/techniques/crossover.py → JobBasedCrossover`



```python
def apply(self, parents):
    offspring = []

    for i in range(0, len(parents) - 1, 2):
        p1 = parents[i].genes
        p2 = parents[i + 1].genes

        jobs = list(set(p1))
        selected_jobs = set(
            random.sample(jobs, random.randint(1, len(jobs) - 1))
        )

        child = [None] * len(p1)

        # Copy selected jobs from parent 1
        for idx, gene in enumerate(p1):
            if gene in selected_jobs:
                child[idx] = gene

        # Fill remaining positions from parent 2
        p2_iter = iter(p2)
        for idx in range(len(child)):
            if child[idx] is None:
                while True:
                    g = next(p2_iter)
                    if child.count(g) < p1.count(g):
                        child[idx] = g
                        break

        offspring.append(Chromosome(child))

    return offspring
```

Randomly selects a subset of jobs from parent 1 and preserves their positions. Remaining genes are filled from parent 2.

1.5 Mutation operators

Mutation as a mechanism to explore local neighborhoods of the search space. For binary encodings, mutation flips bits; for permutations, mutation must preserve feasibility.

1.5.1 Swap mutation

Definition

Swap mutation **exchanges two genes**, producing a small perturbation analogous to bit-flip mutation in binary encodings.

Code implementation

Implemented in:

- `src/ga/techniques/mutation.py → SwapMutation`

```python
def __init__(self, probability=0.1):
    self.probability = probability

def apply(self, offspring):
    for c in offspring:
        if random.random() < self.probability:
            i, j = random.sample(range(len(c.genes)), 2)
            c.genes[i], c.genes[j] = c.genes[j], c.genes[i]
            c.fitness = None
```

1.5.2 Insertion mutation

Definition

Insertion mutation **removes a gene and reinserts it elsewhere**, producing a larger structural change and helping escape local minima.

Code implementation

Implemented in:

- `src/ga/techniques/mutation.py → InsertionMutation`

```python
def apply(self, offspring):
    for c in offspring:
        if random.random() < self.probability:
            i, j = random.sample(range(len(c.genes)), 2)
            gene = c.genes.pop(i)
            c.genes.insert(j, gene)
            c.fitness = None
```

1.6 Elitism

Definition

Elitism copy one (or more) the best individuals to the next generation (ensure high-quality not lost)

Code implementation

Implemented in:

- `src/ga/population.py → replace()`

```
48
49      def replace(self, offspring, elitism=1):
50          elites = self.select_best(elitism)
51          self.chromosomes = elites + offspring[:len(self.chromosomes) - elitism]
52
```

The method preserves the top individual before inserting offspring.

1.7 Convergence and stationary state detection

Definition

Convergence occurs when the population becomes homogeneous and fitness no longer improves. Monitoring fitness evolution is a standard method for detecting stationary states.

Code implementation

Implemented in:

- `src/ga/population.py → is_converged()`

```
def is_converged(self, patience):
    if len(self.history_best_fitness) < patience:
        return False
    recent = self.history_best_fitness[-patience:]
    return all(f == recent[0] for f in recent)
```

The method **checks whether the best fitness has remained constant for a fixed number of generations (`patience`)**, implementing the stationary-state criterion described in the lectures.

2. Experimental results

This section presents the results obtained by executing the Genetic Algorithm on three Job Shop Scheduling Problem instances of increasing size: SMALL (6×6), MEDIUM (10×10), and LARGE (20×20). For each instance, we include a description of the dataset, a comparison of six or more parameter configurations, and an analysis of the best solution's evolution over generations. All experiments were conducted using a population size of 50 and a fixed number of generations per run.

2.1 Small instance

Dataset description

The ft06 dataset is a classical benchmark from the OR-Library, containing 6 jobs and 6 machines. Each job is defined by a sequence of operations, where each operation specifies the required machine and its processing time. The dataset was retrieved from:

`https://people.brunel.ac.uk/%7Emastjjb/jeb/orlib/files/jobshop1.txt`

```
+++++++++++++++++++++++++++++

instance ft06

+++++++++++++++++++++++++++++
Fisher and Thompson 6x6 instance, alternate name (mt06)
6 6
2  1  0  3  1  6  3  7  5  3  4  6
1  8  2  5  4 10  5 10  0 10  3  4
2  5  3  4  5  8  0  9  1  1  4  7
1  5  0  5  2  5  3  3  4  8  5  9
2  9  1  3  4  5  5  4  0  3  3  1
1  3  3  3  5  9  0 10  4  4  2  1
+++++++++++++++++++++++++++++
```

Figure 3. Instance ft06

This instance is suitable for validating the convergence behavior and correctness of the algorithm on small-scale problems.

Results for six parameter configurations

Config	Selection	Crossover	Mutation	Best makespan (fitness)
C1	Tournament	PPX	Swap	64
C2	Tournament	PPX	Insert	61
C3	**Tournament**	**JBX**	**Swap**	**59**
C4	Roulette	PPX	Swap	62
C5	Roulette	PPX	Insert	60
C6	Roulette	JBX	Insert	64

Analysis:

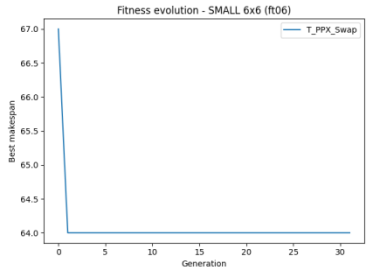
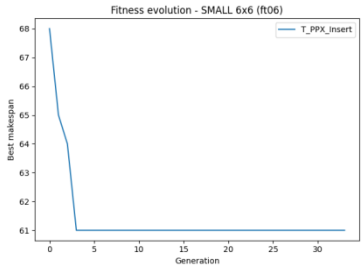
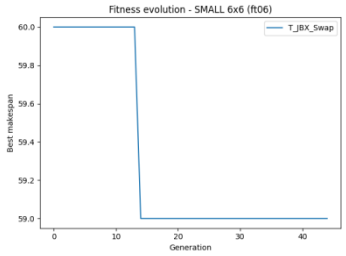
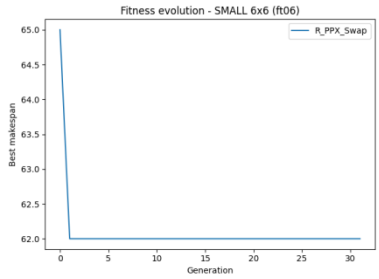
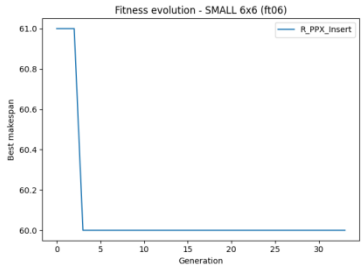
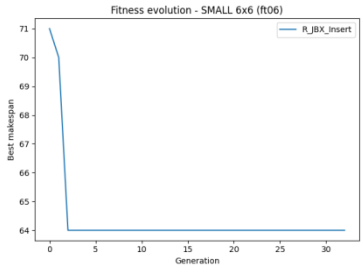
Although the best result in this particular execution (makespan = 59) was obtained using Tournament+JBX+Swap, the **ft06 instance is small and highly sensitive to variation**.

In general, **tournament selection** tends to provide more **stable** convergence due to its controlled selection pressure, while **roulette wheel selection** may **occasionally outperform** it when the initial population happens to contain promising individuals (**as casino games, very random**).

PPX crossover consistently showed **robust** performance, and **JBX** sometimes produced **competitive results** depending on the run.

Swap mutation remained the **most effective** mutation operator, leading to faster convergence than Insertion in most cases.

Evolution of the Best Solution

C1 Tournament_PPX_Swap	C2 Tournament_PPX_Insert	C3 Tournament_JBX_Swap
		
C4 **Roulette_PPX_Swap**	**C5** **Roulette_PPX_Insert**	**C6** **Roulette_JBX_Swap**
		

Each plot shows the best makespan over generations. Configurations using tournament selection converged faster and more effectively. Some roulette-based runs showed no improvement, indicating poor exploration.

Comparative plot

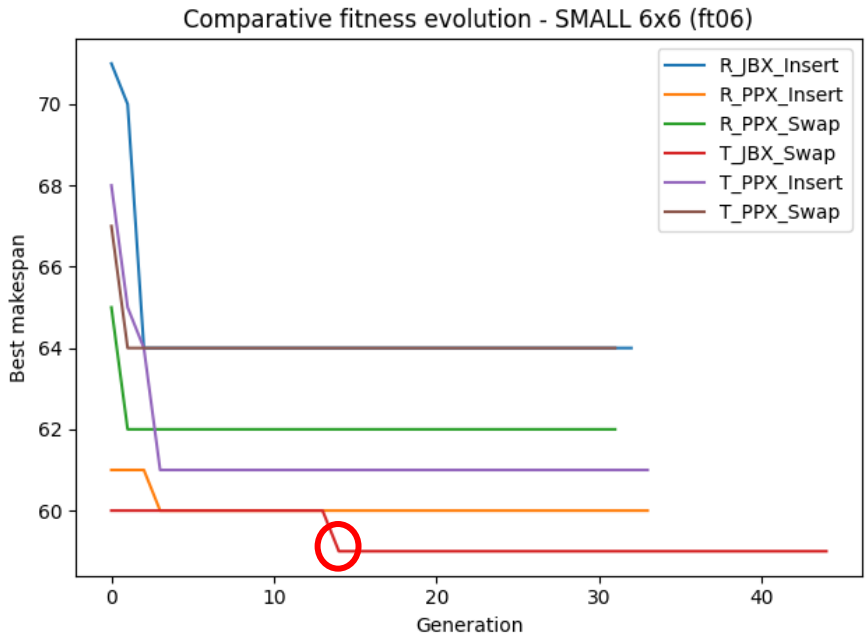


Figure 4. Comparative fitness evolution SMALL

2.2 Medium instance (around 10 machines)

Dataset Description

The la19 dataset contains 10 jobs and 10 machines, with each job defined by a sequence of 10 operations. It was retrieved from the OR-Library and is commonly used to evaluate the scalability of scheduling algorithms.

`https://people.brunel.ac.uk/%7Emastjjb/jeb/orlib/files/jobshop1.txt`

```
instance la19

+++++++++++++++++++++++++++++
Lawrence 10x10 instance (Table 6, instance 4); also called (seta4) or (A4)
10 10
2 44 3  5 5 58 4 97 0  9 7 84 8 77 9 96 1 58 6 89
4 15 7 31 1 87 8 57 0 77 3 85 2 81 5 39 9 73 6 21
9 82 6 22 4 10 3 70 1 49 0 40 8 34 2 48 7 80 5 71
1 91 2 17 7 62 5 75 8 47 4 11 3  7 6 72 9 35 0 55
6 71 1 90 3 75 0 64 2 94 8 15 4 12 7 67 9 20 5 50
7 70 5 93 8 77 2 29 4 58 6 93 3 68 1 57 9  7 0 52
6 87 1 63 4 26 5  6 2 82 3 27 7 56 8 48 9 36 0 95
0 36 5 15 8 41 9 78 3 76 6 84 4 30 7 76 2 36 1  8
5 88 2 81 3 13 6 82 4 54 7 13 8 29 9 40 1 78 0 75
9 88 4 54 6 64 7 32 0 52 2  6 8 54 5 82 3  6 1 26
+++++++++++++++++++++++++++++
```

Figure 5. Instance la19

This instance represents a medium-sized problem with increased combinatorial complexity.

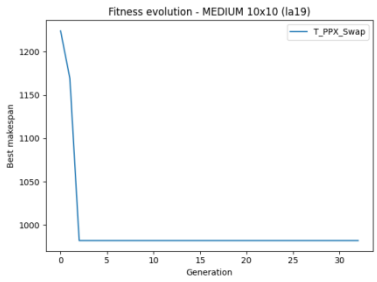
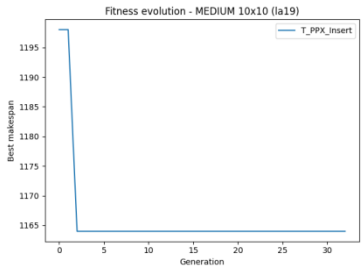
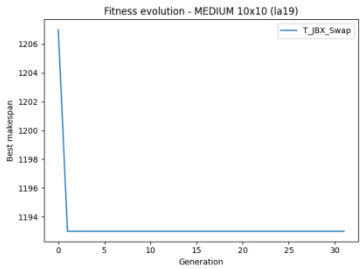
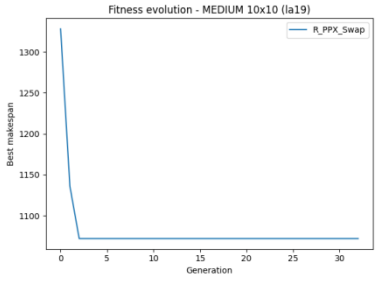
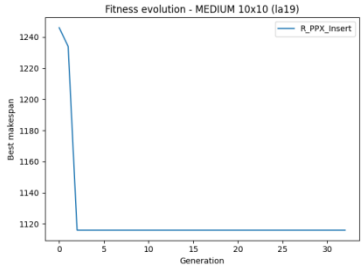
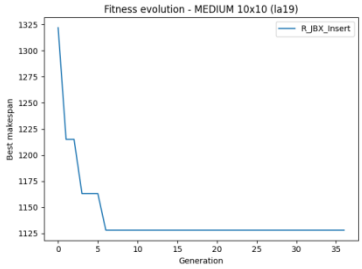
Results for six parameter configurations

Config	Selection	Crossover	Mutation	Best makespan (fitness)
C1	**Tournament**	**PPX**	**Swap**	**982**
C2	Tournament	PPX	Insert	1164
C3	Tournament	JBX	Swap	1193
C4	Roulette	PPX	Swap	1072
C5	Roulette	PPX	Insert	1116
C6	Roulette	JBX	Insert	1128

Analysis:

The best result (makespan = 982) was achieved with Tournament selection, PPX crossover, and Swap mutation. Roulette selection combined with PPX and Swap also performed well (1072), and I think that mutation type is important in medium-scale problems. JBX crossover and roulette selection showed limited improvement.

Evolution of the Best Solution

C1 Tournament_PPX_Swap	C2 Tournament_PPX_Insert	C3 Tournament_JBX_Swap
		
C4 Roulette_PPX_Swap	**C5 Roulette_PPX_Insert**	**C6 Roulette_JBX_Swap**
		

The evolution curves show that most configurations **converge within the first 10–15 generations**. Tournament selection combined with Insertion mutation leads to faster and deeper convergence.

Comparative Plot

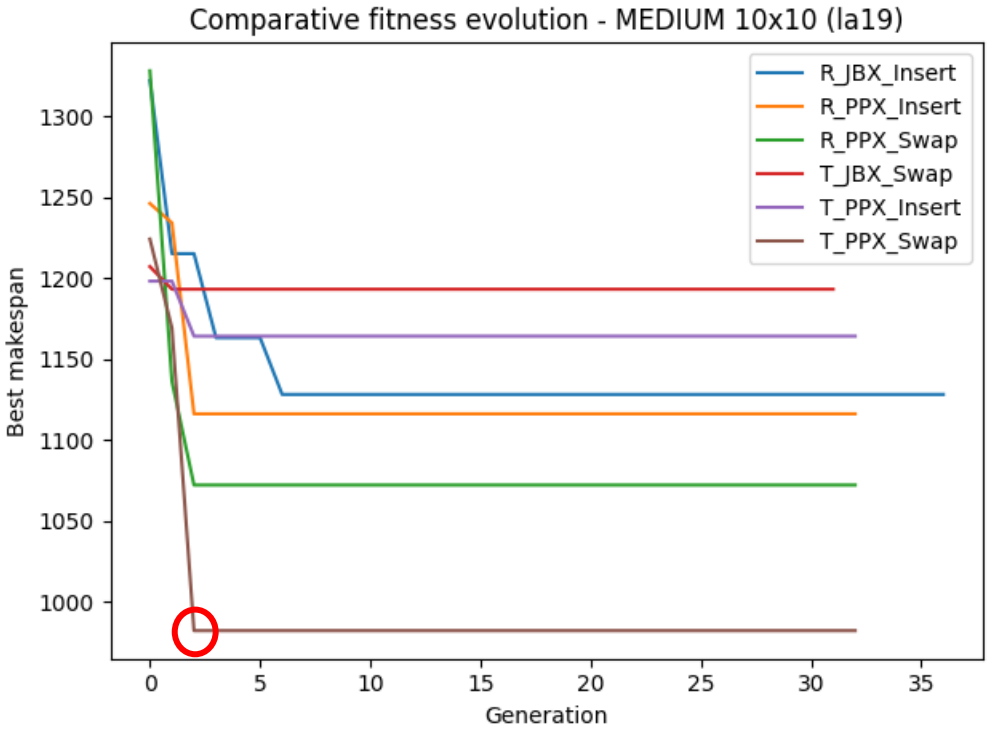


Figure 6. Comparative fitness evolution MEDIUM

2.3 Large instance (15 or more machines)

Dataset Description

The yn1 dataset contains 20 jobs and 20 machines, with each job defined by 20 operations. This instance represents a large-scale scheduling problem with high dimensionality and complex precedence constraints.

`https://people.brunel.ac.uk/%7Emastjjb/jeb/orlib/files/jobshop1.txt`

```
+++++++++++++++++++++++++++++

instance yn1

+++++++++++++++++++++++++++++
Yamada and Nakano 20x20 instance (Table 4, instance 1)
20 20
17 13  2 26 11 35  4 45 12 29 13 21  7 40  0 45  3 16 15 10 18 49 10 43 14 25  8 25  1 40  6 16 19 43  5 48  9 36 16 11
 8 21  6 22 14 15  5 28 10 10  2 46 11 19 19 13 13 18 18 14  3 11  4 21 16 30  1 29  0 16 15 41 17 40 12 38  7 28  9 39
 4 39  3 28  8 32 17 46  0 35 14 14  1 44 10 20 13 12  6 23 18 22  9 15 11 35  7 27 16 26  5 27 15 23  2 27 12 31 19 31
 4 31 10 24  3 34  6 44 18 43 12 32  2 35 15 34 19 21  7 46 13 15  5 10  9 24 14 37 17 38  1 41  8 34  0 32 16 11 11 36
19 45  1 23  5 34  9 23  7 41 16 10 11 40 12 46 14 27  8 13  4 20  2 40 15 28 13 44 17 34 18 21 10 27  0 12  6 37  3 30
13 48  2 34  3 22  7 14 12 22 14 10  8 45 19 38  6 32 16 38 11 16  4 20  0 12  5 40  9 33 17 35  1 32 10 15 15 31 18 49
 9 19  5 33 18 32 16 37 12 28  3 16  2 40 10 37  4 10 11 20  1 17 17 48  6 44 13 29 14 44 15 48  8 21  0 31  7 36 19 43
 9 20  6 43  1 13  5 22  2 33  7 28 16 39 12 16 13 34 17 20 10 47 18 43 19 44  8 29 15 22  4 14 11 28 14 44  0 33  3 28
 7 14 12 40  8 19  0 49 13 11 10 13  9 47 18 22  2 27 17 26  3 47  5 37  6 19 15 43 14 41  1 34 11 21  4 30 19 32 16 45
16 32  7 22 15 30  6 18 18 41 19 34  9 22 11 11 17 29 10 37  4 30  2 25  1 27  0 31 14 16 13 20  3 26 12 14  5 24  8 43
18 22 17 22 12 30 15 31 13 15  4 13 16 47 19 18  6 33  3 30  7 46  2 48 11 42  0 18  1 16  8 25 10 43  5 21  9 27 14 14
 5 48  1 39  2 21 18 18 13 20  0 28 15 20  8 36  6 24  9 35  7 22 19 36  3 39 14 34  4 49 17 36 11 38 10 46 12 44 16 13
14 26  1 32  2 11 15 10  9 41 13 10  6 26 19 26 12 13 11 35  5 22  0 11  7 24 17 33  8 11 10 34 16 11  3 22  4 12 18 17
16 39 10 24 17 43 14 28  3 49 15 34 18 46 13 29  6 31 11 40  7 24  1 47  9 15  2 26  8 40 12 46  5 18 19 16  4 14  0 21
11 41 19 26 16 14  3 47  0 49  5 16 17 31  9 43 15 20 10 25 14 10 13 49  8 32  6 36  7 19  4 23  2 20 18 15 12 34  1 33
11 37  5 48 10 31  7 42  2 24  1 13  9 30 15 24  0 19 13 34 19 35  8 42  3 10 14 40  4 39  6 42 12 38 16 12 18 27 17 40
14 19  1 27  8 39 12 41  5 45 11 40 10 46  6 48  7 37  3 30 17 31  4 16 18 29 15 44  0 41 16 35 13 47  9 21  2 10 19 48
18 38  0 27 13 32  9 30  7 17 14 21  1 14  4 37 17 15 16 31  5 27 10 25 15 41 11 48  3 48  6 36  2 30 12 45  8 26 19 17
 1 17 10 40  9 16  5 36  4 34 16 47 19 14  0 24 18 10  6 14 13 14  3 30 12 23  2 37 17 11 11 23  8 40 15 15 14 10  7 46
14 37 10 28 13 13  0 28  2 18  1 43 16 46  8 39  3 30 12 15 11 38 17 38 18 45 19 44  9 16 15 29  5 33  6 20  7 35  4 34
+++++++++++++++++++++++++++++
```

Figure 7. Instance yn1

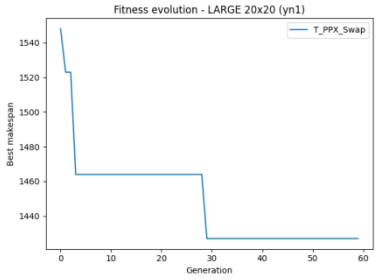
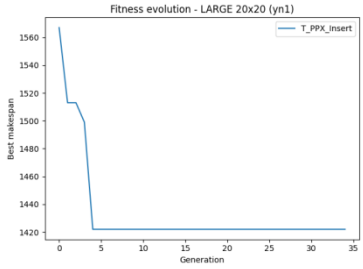
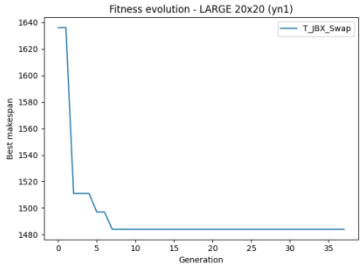
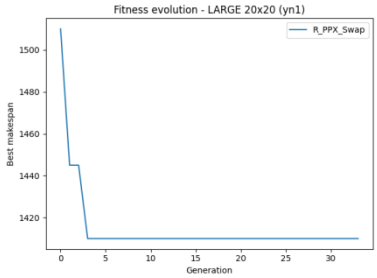
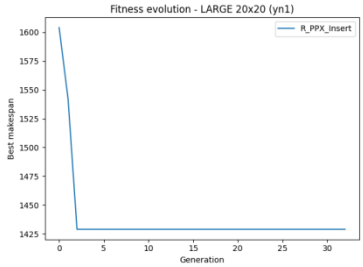
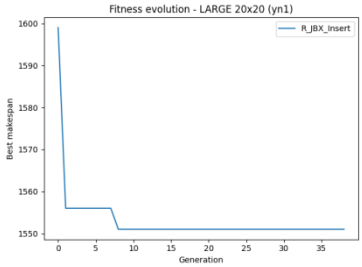
Results for six parameter configurations

Config	Selection	Crossover	Mutation	Best makespan (fitness)
C1	Tournament	PPX	Swap	1427
C2	Tournament	PPX	Insert	1422
C3	Tournament	JBX	Swap	1484
C4	**Roulette**	**PPX**	**Swap**	**1410**
C5	Roulette	PPX	Insert	1429
C6	Roulette	JBX	Insert	1551

Analysis:

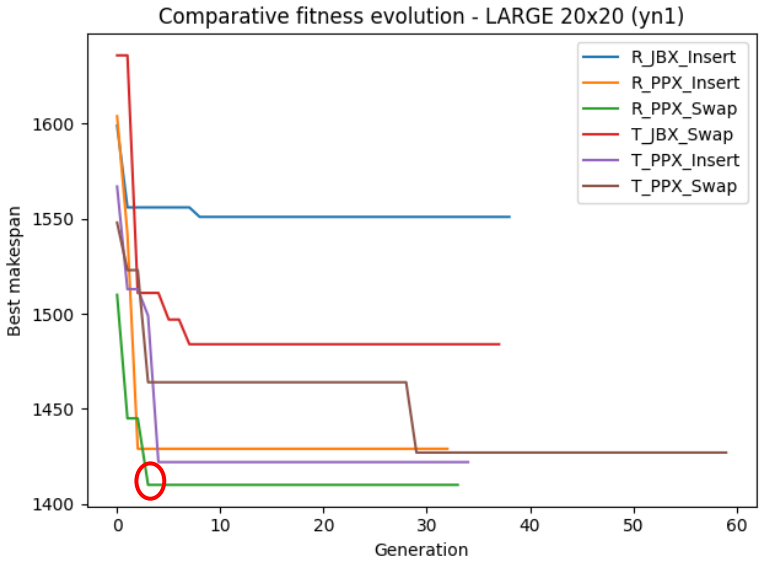
The results obtained for the large-scale instance (20×20) shows important information about the behavior of the Genetic Algorithm under different operator configurations. The best makespan achieved across all runs was **1410**, obtained using roulette selection, PPX crossover, and Swap mutation (configuration C4). This is a notable difference from the SMALL and MEDIUM instances, where tournament selection tended to dominate..

Evolution of the Best Solution

C1 Tournament_PPX_Swap	C2 Tournament_PPX_Insert	C3 Tournament_JBX_Swap
		
C4 **Roulette_PPX_Swap**	**C5** **Roulette_PPX_Insert**	**C6** **Roulette_JBX_Swap**
		

Most configurations converge within 10–15 generations. The best-performing configuration (R_PPX_Swap) shows a steep drop in the first few generations and stabilizes early.

Comparative Plot



2.4 Conclusions

The results obtained across the three instances confirm that the way the problem is encoded, validated, and evaluated has a direct impact on how the Genetic Algorithm behaves.

The choice of techniques (selection, crossover, and mutation) affects the results and also changes depending on the size of the instance.

SMALL instance (6x6)

In the SMALL instance, several configurations reached similar makespans, and the **best result changed between runs**. This shows that the search space is small enough for different operator combinations to work well, **and randomness plays a significant role**. Even so, Tournament selection tended to converge more steadily, while Roulette occasionally matched it when the initial population was favourable, but very randomly.

MEDIUM instance (10x10)

In the MEDIUM instance, the algorithm behaved **more predictably**. Tournament selection combined with PPX crossover and Swap mutation produced the best results, showing that as the problem grows, the algorithm benefits from techniques that preserve structure and guide the search more consistently. The PPX crossover proved especially effective at maintaining valid operation sequences, which is essential given the codification used.

LARGE instance (20x20)

The LARGE instance keeps diversity. Roulette selection achieved the best makespan when paired with PPX and Swap. In very large search spaces, too much selection pressure (as in Tournament) can lead to premature convergence, while **roulette's stochastic nature helps explore more of the solution space before settling**. PPX remained the most reliable crossover technique, and Swap mutation continued to introduce small but useful changes without breaking feasibility.

The experiments show that there is no single "best" configuration for every case, but the codification and validation strategy used in the project works well across all sizes. PPX crossover and Swap mutation stand out as the most robust techniques, while the best selection method depends on the scale of the problem: tournament is more effective in small and medium instances, whereas roulette becomes advantageous in large ones (but ramdom). These results reinforce the importance of adapting the GA's components to the characteristics of the problem and the structure of the chromosome.

3. Second optimization method (optional)

A second optimization method was implemented: **Simulated Annealing (SA)**. I tried to reuse the same problem encoding, the same makespan evaluation, and the same instance structure already used in the GA.

3.1 SA components

Simulated Annealing works with exactly the same elements as the GA:

- **Same chromosome representation:** A solution is a list of job IDs, one per operation.
- **Same fitness function:** SA use the same `compute_makespan` function used in the GA, so results are directly comparable.
- **Same instance structure:** The `JobShopInstance` class is reused without changes.
- **Initial solution from the GA:** SA starts from the best chromosome of an initial GA population

3.2 Simulated Annealing

Simulated Annealing explores one solution at a time:

1. Start from a valid solution (taken from the GA).
2. Create a small variation of it (a **neighbor**) by swapping two positions in the chromosome.
3. Evaluate the new solution using the same makespan function.
4. If the new solution is better, keep it.
5. If it is worse, sometimes keep it anyway, depending on a “temperature” value that decreases over time.
6. Repeat until the temperature is very low or the iteration limit is reached.

3.3 Parameters

The following parameters were used:

- **Initial temperature:** 1000
- **Final temperature:** 1
- **Cooling rate:** 0.95
- **Max iterations:** 2000
- **Neighbor operator:** swap between two random positions

3.4 Comparison with the Genetic Algorithm

Both algorithms use the same encoding and fitness function, so their results can be compared directly.

In general:

- SA is faster because it works with a single solution instead of a population.
- It depends more on the initial solution, which is why starting from the GA's best individual helps.
- The GA tends to perform better on larger instances (population).