# 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

### 1.1.1 Definition

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

The JSSP belongs to this category. 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 to explore different interleavings across jobs.

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

### 1.1.2 Code implementation

The 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” step described in the lectures.

## 1.2 Fitness Evaluation

### 1.2.1 Definition

The course defines fitness as a non-negative value that quantifies the quality of an individual. Although the examples focus on maximization, the lectures clarify that GAs can solve minimization problems by transforming the fitness or adapting the selection operator.

### 1.2.2 Code implementation

The fitness function is the **makespan**, computed by simulating the schedule. This corresponds to the "Evaluate fitness of all individuals" step in the GA pseudocode presented in the lectures.

The simulation is implemented in:

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

Since roulette wheel selection assumes maximization, the implementation applies a fitness inversion:

adjusted_fitnessi=(fmax□−fi)+ε  (fix format)

This adaptation is consistent with the course's requirement that roulette wheel selection must operate on non-negative, maximization-oriented fitness values.

## 1.3 Selection Operators

The course presents four selection schemes: roulette wheel, rank selection, tournament selection, and stochastic universal sampling. Two of these are implemented in this project.

### 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 of roulette wheel selection.

These properties are emphasized in the lectures as advantages for maintaining exploration.

#### *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. The lectures highlight:

- 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`

The code adapts the method for minimization by inverting fitness values, ensuring compatibility with the theoretical model taught in class.

## 1.4 Crossover Operators

The course introduces classical crossover operators (one-point, two-point, uniform) for binary encodings, but also states that for non-binary or permutation encodings, crossover must be adapted to preserve feasibility.

This project implements two such specialized operators.

### 1.4.1 Precedence Preserving Crossover (PPX)

#### *Definition*

For permutation-based problems, crossover must:

- preserve the number of occurrences of each gene,
- maintain structural constraints,
- avoid producing infeasible offspring.

These principles are consistent with the generalization guidelines provided in the lectures.

#### *Code implementation*

Implemented in:

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

The operator selects genes alternately from the parents while removing them from both lists, ensuring feasibility without repair.

### 1.4.2 Job-Based Crossover (JBX)

#### *Definition*

This operator 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`

## 1.5 Mutation Operators

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

Two permutation-safe mutation operators are implemented.

### 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`

### 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`

## 1.6 Elitism

### Definition

The lectures describe elitism as copying the best individuals to the next generation to ensure that high-quality solutions are not lost. This improves convergence stability.

### Code implementation

Implemented in:

- `src/ga/population.py → replace()`

The method preserves the top individual before inserting offspring.

## 1.7 Convergence and Stationary State Detection

### Definition

The course explains that 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:

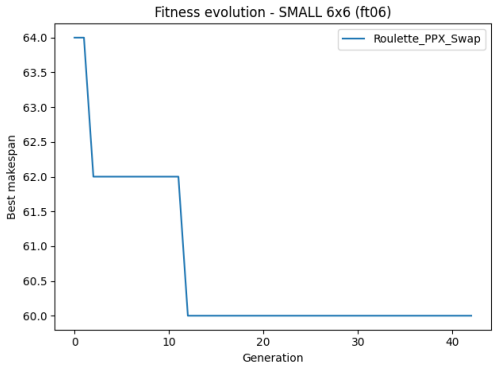
- `src/ga/population.py → is_converged()`

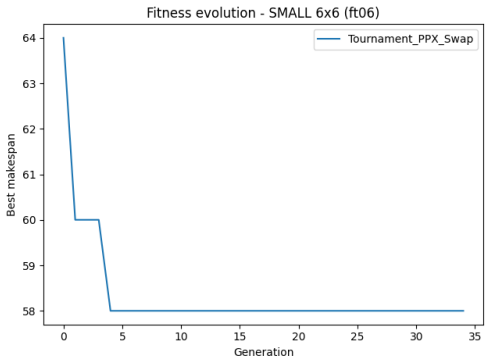
The method checks whether the best fitness has remained constant for a fixed number of generations (`patience`), implementing the stationary-state criterion described in the lectures.

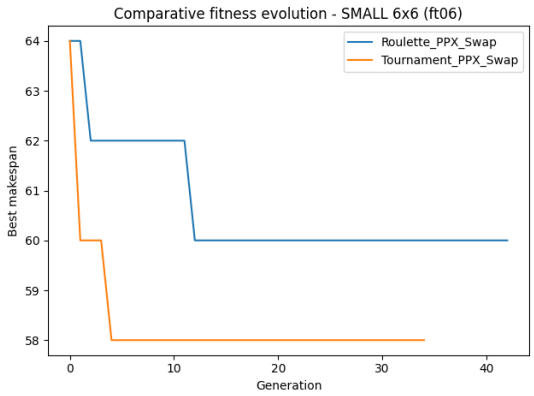
# 2. Experimental Results (pending to plot all experiments to explain single and comparative)

## 2.1 Dataset 1: Small Instance (3–5 Machines)

---




## 2.2 Dataset 2: Medium Instance (~10 Machines)

---

## 2.3 Dataset 3: Large Instance (15+ Machines)

---

# Conclusion