ACTIVITY 2

Optimization with Genetic Algorithms
Job Shop Scheduling Problem

(by Victoria Joven)

---

https://github.com/victoriajoven/NeuronalActivity_2

1. Chromosome Description and Algorithm Design

Code structure reference: `src/ga/`, `src/jobmanager/`, `src/evaluation/`

1.1 Chromosome Representation

Implemented in: `src/ga/chromosome.py`

Each solution is encoded as a **chromosome representing a job-based operation sequence**. The chromosome is implemented as a linear list of genes, where each gene corresponds to a **job identifier**. Each job identifier appears as many times as the number of operations of that job.

In the code, the `Chromosome` class encapsulates this representation:

- `genes` stores the ordered list of job identifiers.
- `validate()` ensures structural correctness of the chromosome before it is used by GA operators.
- `copy()` is used during reproduction to duplicate individuals safely.

This representation guarantees that:

- All operations of every job are scheduled exactly once.
- Job precedence constraints are preserved implicitly during decoding.

A chromosome is considered valid if:

- The gene list is not empty and has a minimum length.
- Genes are stored in a list or tuple structure.
- All genes correspond to valid job identifiers.

1.2 Decoding and Schedule Construction

Implemented in: `src/jobmanager/`

Chromosomes are decoded using a **schedule construction procedure** that simulates job execution step by step. This logic is implemented in the job manager module, which:

- Tracks the next pending operation of each job.
- Tracks machine availability times.
- Assigns operations respecting both job order and machine constraints.

For each gene in the chromosome, the next operation of the corresponding job is scheduled on its required machine at the earliest feasible time.

The decoding process produces a complete schedule and its corresponding **makespan**, which is later used as the fitness value.

1.3 Fitness

Implemented in: `src/evaluation/`

The fitness of a chromosome is defined as the **total execution time (makespan)** of the generated schedule. Fitness evaluation is separated from the genetic operators and handled by the evaluation module.

The evaluation logic:

- Invokes the schedule builder.
- Computes the completion time of the last operation.
- Assigns this value to `chromosome.fitness`.

All problem constraints are enforced during schedule construction, ensuring that:

- Machines execute a single operation at a time.
- Operations are non-preemptive.
- Job operation order is strictly respected.

1.4 Selection techniques

Implemented in: `src/ga/techniques/selection/`

The following selection techniques are implemented as interchangeable components:

1. **Tournament Selection**
    - Randomly samples a subset of the population.
    - Selects the individual with the best fitness.
2. **Roulette Wheel Selection**
    - Computes selection probabilities based on inverse makespan.
    - Selects individuals probabilistically.

The selection strategy is configurable and used during each generation to choose parents for reproduction.

1.5 Crossover techniques

Implemented in: `src/ga/techniques/crossover/`

Two crossover strategies compatible with permutation-based chromosomes are implemented:

1. **Order Crossover (OX)**
    - Selects a random subsequence from one parent.
    - Preserves relative ordering from the second parent.
2. **Position-Based Crossover**
    - Randomly selects gene positions to inherit from one parent.
    - Fills remaining positions from the second parent while preserving order.

Both operators ensure that offspring chromosomes remain valid and contain the correct number of job identifiers.

1.6 Mutation techniques

Implemented in: `src/ga/techniques/mutation/`

Multiple mutation techniques are implemented to maintain population diversity:

1. **Swap Mutation**
    - Randomly selects two positions and swaps their genes.
2. **Insertion Mutation**
    - Removes a gene from one position and reinserts it at another.

Mutation operators are applied probabilistically and preserve chromosome validity.

1.7 Elitism

Status in code: *Not explicitly implemented*

The current implementation does **not include an explicit elitism mechanism**. Individuals are selected exclusively through the configured selection operators, and no chromosome is forcefully preserved between generations.

As a consequence, although high-quality solutions have a higher probability of being selected, there is no strict guarantee that the best individual of a generation survives unchanged into the next one.

1.8 Population Size and Stopping Criteria

Implemented in: `src/experiments/` and `main.py`

Population size and genetic algorithm parameters are defined explicitly in the experiment configuration and passed to the GA at execution time.

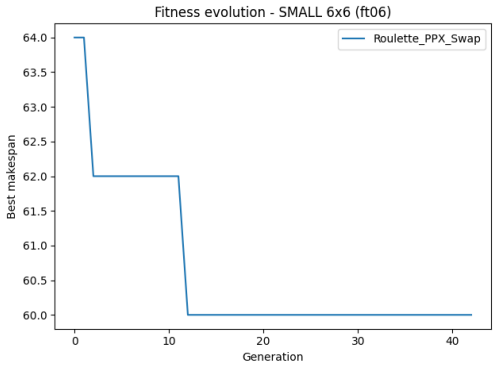
The stopping criterion of the algorithm is **a fixed maximum number of generations**. The genetic algorithm runs for the predefined number of iterations without an explicit convergence or stagnation-based early stopping mechanism.

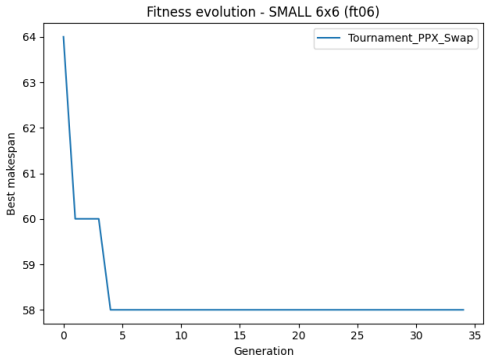
Convergence and stationary behavior are therefore **analyzed later**, by inspecting the evolution of the best fitness value across generations, which is recorded during execution and later visualized in the results section.

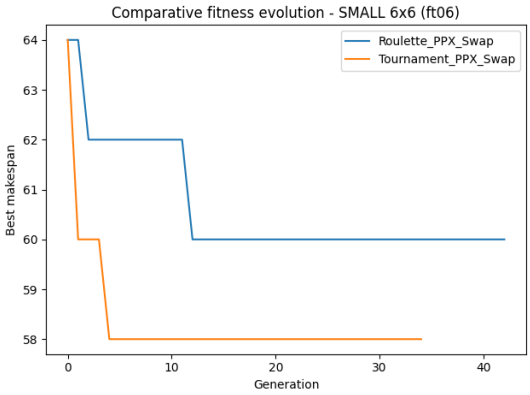
---

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