

Assignment: Genetic Algorithm (GA) — Object-Oriented vs Functional Programming

Overview

In this assignment, you will implement a Genetic Algorithm (GA) twice: once using Object-Oriented Programming (OOP) principles and once using Functional Programming (FP) principles. This assignment helps you understand both paradigms while learning a foundational optimization method used in artificial intelligence, operations research, and evolutionary computation.

Learning Objectives

By completing this assignment, you will be able to:

1. Explain the core concepts of Genetic Algorithms.
2. Implement a GA from scratch to solve standard optimization problems.
3. Apply OOP design patterns such as abstraction, encapsulation, and modularity.
4. Apply FP principles such as immutability, pure functions, and function composition.
5. Compare design trade-offs between OOP and FP paradigms.

Background: What Is a Genetic Algorithm?

A Genetic Algorithm (GA) is a population-based search and optimization technique inspired by the process of natural selection. It simulates evolution by maintaining a population of candidate solutions that evolve over generations toward better solutions.

1. Representation

Each solution is represented as a chromosome, often encoded as a bitstring (e.g., 1010110010). Each bit corresponds to a decision variable.

2. Population

A set of chromosomes forms a population. The algorithm evaluates how good each chromosome is using a fitness function.

3. Fitness Function

The fitness measures the quality of a solution. Higher fitness means a better solution. For example, in the OneMax problem, the fitness is simply the number of 1s in the chromosome.

4. Selection

Chromosomes are selected for reproduction based on fitness — better ones have a higher chance of being selected. A common method is tournament selection, where a few individuals are chosen randomly, and the best among them becomes a parent.

5. Crossover (Recombination)

Two parent chromosomes exchange parts of their bitstrings to form offspring. This simulates genetic recombination and introduces diversity.

Example (one-point crossover):

```
Parent1: 11001|0110
Parent2: 00111|1001
Offspring1: 11001|1001
Offspring2: 00111|0110
```

6. Mutation

Each bit in the chromosome has a small probability of flipping ($0 \rightarrow 1$ or $1 \rightarrow 0$). Mutation prevents the algorithm from getting stuck in local optima.

7. Elitism and Replacement

The next generation is formed from the offspring, sometimes keeping a few of the best individuals unchanged (elitism). This ensures that the best solutions are not lost.

8. Termination

The algorithm runs for a fixed number of generations or until a stopping criterion is met (e.g., reaching maximum fitness).

Problems to Solve

You will implement your GA to solve two classic problems:

1. OneMax Problem

- Goal: Maximize the number of 1s in a binary string of length $L = 100$.
- Fitness: Number of 1s in the chromosome.

2. 0/1 Knapsack Problem

- Given: $n = 100$ items with random values and weights.
- Capacity: $C = 40\%$ of the total weight.
- Fitness: Total value of selected items, but fitness = 0 if total weight exceeds capacity.

Both problems will use the same GA structure (same operators and parameters) — only the fitness function differs.

Genetic Algorithm Configuration

Parameter	Description	Value
Representation	Bitstring	—
Population size	Number of individuals	100
Parent selection	Tournament ($k=3$)	—
Crossover	One-point crossover	Probability = 0.9
Mutation	Bit-flip mutation	Probability per bit = $1/L$
Replacement	Generational with elitism	$e = 2$
Termination	Fixed number of generations	300

Use the same random seed (42) for both implementations to ensure reproducibility.

Implementation Requirements

You must create two independent implementations:

A. OOP Implementation

Use classes and objects to represent GA components.

Your design must include at least the following abstractions:

- Chromosome class (represents a candidate solution)
- SelectionStrategy (interface or abstract class)
- CrossoverStrategy
- MutationStrategy
- Population class
- GeneticAlgorithm class (coordinates the process)

All state must be properly encapsulated, and the code should follow good OOP principles.

B. FP Implementation

Implement the GA using pure functions and immutable data structures.

Requirements:

- No global or mutable state.
- No classes.
- Functions should not have side effects.
- Use higher-order functions like `map`, `filter`, and `reduce` where applicable.

Both implementations must solve both the OneMax and Knapsack problems using identical parameters and produce comparable results.

Deliverables

Your submission must include the following structure:

```
ga-assignment/
  README.md
  oop/
    src/
    tests/
    run.py
  fp/
    src/
    tests/
    run.py
  reports/
    onemax_curve.png
    knapsack_curve.png
    results_oop.json
    results_fp.json
```

README.md

Include:

- Your name and student ID.
- Instructions to run both versions.
- Explanation of your OOP and FP designs.
- Reflection (≤500 words) comparing both paradigms.

Output

Each version should output:

- Fitness evolution over generations (as JSON and plots).
- Final best fitness and runtime.

Testing

Include minimal unit tests covering fitness evaluation, selection, crossover, mutation, and improvement over generations.

Grading Rubric (100 pts)

Category	Description	Points
GA Correctness	Correct implementation and convergence	30
OOP Design	Encapsulation, modularity, clarity	20
FP Design	Purity, immutability, composition	20
Reporting	Curves, JSON outputs, reproducibility	15
Testing	Unit tests for key operators	10
Code Quality	Readability and structure	5
Bonus	Extensible design or extra analysis	+5

Submission

- Submit your full project folder as a .zip or Git repository link.
- Ensure both implementations run correctly using `python oop/run.py` and `python fp/run.py`.
- Deadline: 15/12/2025