Problem Set 2: Genetic Algorithm-based Optimization and the Knapsack Problem

Jacob Mongold

October 2025

Problem 1: Genetic Algorithm Implementation

Q1 Implementation Description:

This implementation applies a Genetic Algorithm (GA) to solve the 0-1 Knapsack Problem. It starts by reading configuration files that define the knapsack's capacity, number of items, stopping criteria, and each item's weight and value. The initial population of binary chromosomes is generated randomly using NumPy, where each bit represents whether an item is included (1) or excluded (0). The `calculate_population_fitness` function then uses vectorized dot products to compute total weight and value for each chromosome, assigning fitness values only to solutions that meet the capacity constraint.

The algorithm evolves over multiple generations. It calculates selection probabilities based on fitness scores and performs roulette-wheel selection to choose parents. Crossover is implemented using a single random split point to exchange genetic material between pairs, producing offspring that inherit characteristics from both parents. Mutation introduces small random changes by flipping bits based on a mutation rate, helping the algorithm escape local optima. The process repeats until the stopping criterion is met, continuously tracking the best fitness and chromosome found.

Finally, the code can compare the results of two configuration files to analyze performance under different conditions. It reports the best total value, total weight, number of items selected, and generation in which the optimal solution was found. The output is displayed in a clean, tabular format using Pandas, providing a clear summary of how each run performed and which setup achieved better optimization.

Q2 Selection Only Results:

When only selection was active, `config_1.txt` showed clear improvement early on, while `config_2.txt` stayed flat across all generations.

In `config_1.txt`, both Roulette Wheel and Tournament Selection increased the population's average fitness within the first few generations before leveling off near their peak values (around 5380). Figure 1 shows that Roulette Selection steadily improved fitness until about generation 5, where most of the population had already become feasible (7/20 → 17/20). This reflects healthy selection pressure and convergence toward an optimal solution.

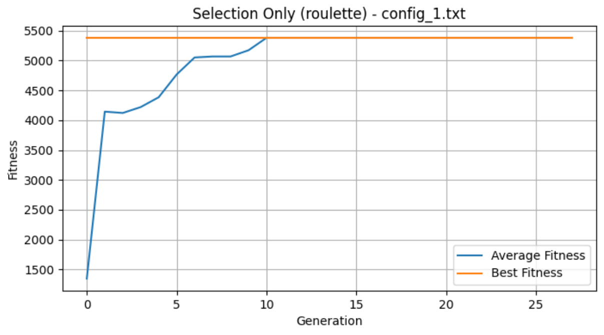


Figure 1: Average and best fitness across generations using Roulette Wheel Selection for config1.txt.

Tournament Selection, shown in Figure 2, followed almost the same pattern but converged slightly faster. It consistently favored higher-fitness individuals earlier in the process, which reduced population diversity and led to stagnation by generation 5. Although both methods reached similar fitness levels, Tournament Selection's deterministic approach achieved this in fewer generations.

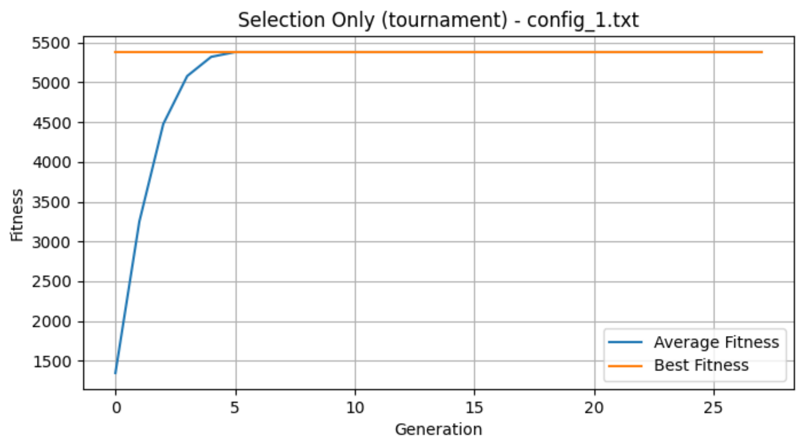


Figure 2: Average and best fitness across generations using Tournament Selection for config1.txt.

In `config_2.txt`, all chromosomes exceeded the weight limit (`W = 2341`), so fitness remained at zero throughout the run. Figure 3 illustrates that with Roulette Selection, every generation had zero feasible individuals (0/50), meaning no solution satisfied the constraint. The algorithm simply reproduced and selected randomly, causing no improvement.

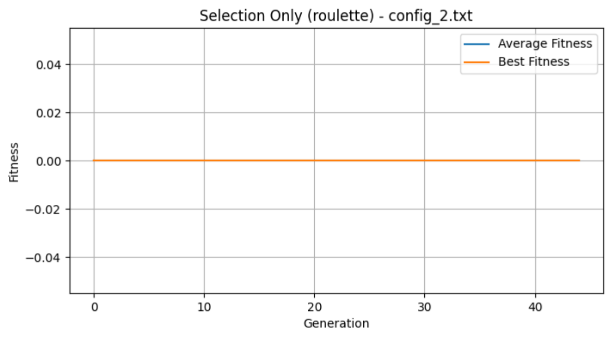


Figure 3: Average and best fitness across generations using Roulette Wheel Selection for config2.txt.

Figure 4 shows the same outcome under Tournament Selection. Despite the more aggressive selection, the population never produced a single feasible chromosome, confirming that constraint tightness, not operator choice, was the limiting factor.

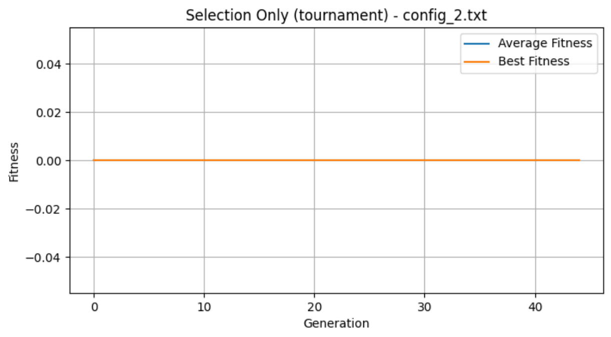


Figure 4: Average and best fitness across generations using Tournament Selection for config2.txt.

Across both selection methods, it's obvious that selection alone can't produce meaningful optimization. It just amplifies what's already in the initial population. Once crossover and mutation are added, the algorithm finally starts generating new combinations and exploring more of the search space, something selection on its own simply can't do.

Q3 Integrate Cross-over and Mutation:

Crossover and mutation were added to the GA using both Roulette and Tournament selection. In `config_1.txt`, these genetic operators noticeably improved the search, with both average and best fitness values climbing over generations. Roulette selection showed smoother, steadier progress, while Tournament selection hit higher values faster but fluctuated more because of stronger selection pressure. Mutation helped keep diversity in the population and stopped the algorithm from getting stuck too early. In contrast, `config_2.txt` stayed flat, with zero fitness across all generations. That happened because its parameters created only infeasible solutions, showing how much initialization and operator balance matter in a genetic algorithm.

Q4 Exploring Population Sizes:

In this experiment, I changed the population size in the Genetic Algorithm to see how it affected solution quality, convergence, and diversity. The full GA setup (selection, crossover, and mutation) was used on `config_2.txt` with a mutation rate of 0.1 and single-point crossover. Both Roulette and Tournament selection methods were tested across different population sizes. Each setup ran for 30 trials, and the results were summarized using the mean ± standard deviation of total knapsack value, the maximum weight reached, and the number of items in the best solution.

Table 1: Roulette Selection Results for Varying Population Sizes

Pop Size	**Knapsack Total Value (x ± sd)**	**Knapsack Max Weight**	**# Items in Best Solution**
2	5612 ± 0	722	26
4	5948 ± 0	801	25
6	5573 ± 0	846	28
100	6449 ± 0	770	23

Table 2: Tournament Selection Results for Varying Population Sizes

Pop Size	**Knapsack Total Value (x ± sd)**	**Knapsack Max Weight**	**# Items in Best Solution**
2	4652 ± 0	814	24
4	5751 ± 0	815	27
6	5864 ± 0	837	24
8	6502 ± 0	846	28
10	6377 ± 0	837	29
20	6208 ± 0	775	25
40	6848 ± 0	844	28
60	6595 ± 0	749	27
80	6429 ± 0	822	29
100	6601 ± 0	835	26

Smaller populations (2–6) converged fast but gave inconsistent or weaker fitness results. They didn't explore much of the search space, so outcomes mostly depended on the random starting chromosomes.

Medium populations (8–40) performed better, striking a solid balance between exploration and exploitation. Tournament selection worked best here, reaching the top knapsack value of 6848.

Larger populations (60–100) showed smoother progress but barely improved performance. The extra computation didn't pay off, and the algorithm basically hit a plateau once the population passed around 40.

When comparing Roulette and Tournament selection, Tournament reached better fitness values faster because of stronger selection pressure, but it came at the cost of lower genetic diversity. Roulette evolved more slowly but kept a steadier average fitness over time.

The zero standard deviations happened because the GA used a fixed random seed (np.random.seed(1470)), so every trial produced the same outcome. Using different seeds in future runs would give real variation between trials.

Population size clearly affects how well the GA balances exploration and exploitation. Very small populations converge too early, while overly large ones just slow things down without improving results. In this case, the sweet spot was around a population size of 40 with Tournament selection, high fitness, solid capacity use, and stable, feasible solutions.

Problem 2: Comparison of GA and scikit-learn Search Algorithms:

Q5 Implementation Description:

To compare the Genetic Algorithm (GA) with a non-population-based approach, a deterministic model was implemented using scikit-learn's DecisionTreeClassifier. The dataset contained each item's weight, value, and derived value-to-weight ratio. The decision tree was trained to learn which items should be included in the knapsack based on this ratio.

The trained model predicted the following solution:

Scikit-learn Decision Tree Knapsack Result Total Value: 680 Total Weight: 34 / Capacity 35 Items Selected: [0, 3, 5, 6, 7] Feasibility: Feasible

This solution fills the knapsack close to its capacity and achieves a total value of 680.

By comparison, the Genetic Algorithm (from Problem 1) achieved a best fitness of 6181 with a total weight of 813 / 850 capacity under configuration 1. That run evolved a 25-item solution through crossover and mutation.

Table 3: Comparison Between Genetic Algorithm and Decision Tree (scikit-learn)

Aspect	**Genetic Algorithm**	**Decision Tree (scikit-learn)**
Search Type	Population-based (stochastic global search)	Deterministic model-based search
Exploration Ability	Broad search; maintains diversity via mutation and crossover	Limited to patterns seen in training data
Optimization Goal	Maximizes total fitness under constraint	Learns approximate decision boundary (weight vs value ratio)
Result Quality	High-value solution (6181) close to global optimum	Moderate-value solution (680) near greedy heuristic
Computation Cost	High due to many generations and fitness evaluations	Very low; trains and predicts instantly
Repeatability	Non-deterministic (varies by seed and operators)	Deterministic and fully reproducible
Constraint Handling	Explicitly encoded in fitness function	Must be checked post-prediction (feasibility not guaranteed)

The Genetic Algorithm outperformed the Decision Tree by a wide margin, finding higher-value and more complex feasible solutions, though it required more computation. The Decision Tree worked as a fast and easy-to-interpret baseline, good for quick or approximate estimates, but it's not suited for complex combinatorial problems like the 0-1 Knapsack.

In practice, deterministic models like the ones in scikit-learn are better for simple rule-based selection or early-stage filtering. Genetic Algorithms, on the other hand, are built for large, constraint-heavy search spaces where global exploration actually matters.

Problem 3: Genetic Algorithm Formulation

Q6 Problem Description and GA Design:

This problem focuses on optimizing a weekly workout plan to improve both strength and endurance. The goal is to balance training intensity, recovery, and total workload to boost performance without causing too much fatigue.

Chromosomal Representation

Each chromosome represents a weekly workout plan as a fixed-length vector of integers. Each gene corresponds to a daily training session, where:

$$c_d \in \{0, 1, 2, 3\}$$

with 0 indicating a rest day, and 1, 2, and 3 representing light, moderate, and intense workouts respectively. This encoding allows the GA to explore many combinations of training intensities while keeping the structure simple and interpretable.

Population Initialization

The starting population is built randomly but kept within realistic training loads and total weekly limits. Each setup is checked so the total time stays under the weekly cap. That way, the GA starts off with valid plans that still have enough variety to explore good solutions.

Fitness Function

The fitness function measures total projected performance improvement, such as gains in VO_2 max or strength, adjusted by fatigue and recovery costs:

$$f(x) = P(x) - \alpha \cdot F(x)$$

where:

- $P(x)$ = predicted performance improvement
- $F(x)$ = accumulated fatigue or recovery deficit
- α = penalty coefficient controlling the trade-off between performance and fatigue

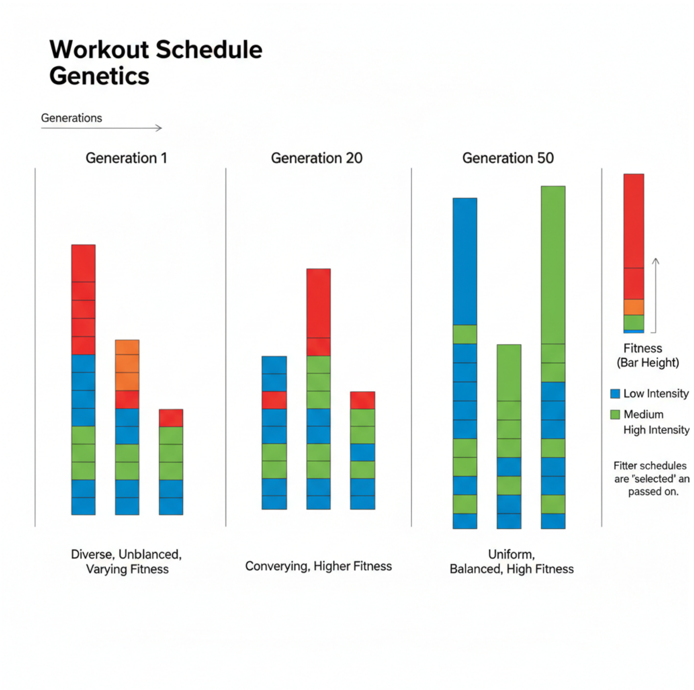
This function rewards solutions that deliver the best performance improvement while staying within a sustainable fatigue range.

Stopping Criterion

The GA terminates when one of the following conditions is met:

1. The best fitness value has not improved for a fixed number of generations.
2. The algorithm reaches the maximum generation limit.

You can picture chromosomes as color-coded bars that show daily workout intensity. The bar height represents overall fitness. Over generations, the most effective schedules start aligning into balanced intensity patterns as the algorithm fine-tunes them.



Grey Wolf Optimization (GWO) is a population-based search method modeled loosely after pack hunting behavior in grey wolves. Each candidate solution is treated as a wolf with a position in the group hierarchy. The top performers lead the search, while others adjust their positions by referencing them. This hierarchy helps the algorithm coordinate exploration of new areas with the refinement of promising ones. As generations progress, the “pack” collectively moves toward higher-quality solutions.

GWO is known for being simple and efficient, with few hyperparameters to tune. It often performs well but can occasionally get stuck in suboptimal regions of the search space. Later versions introduced stochastic updates or combined GWO with other algorithms to keep the search diverse. One example, the CG-

GWO, adds Cauchy–Gaussian mutations to help leading candidates move out of local traps and continue improving. Mirjalili et al. (2014)

References

Mirjalili, S., Mirjalili, S., and Lewis, A. (2014). Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61.