COMPUTATIONAL INTELLIGENCE FOR OPTIMIZATION PROJECT

Master in Data Science and Advanced Analytics

NOVA Information Management School

Universidade Nova de Lisboa

Sports League Optimization

Ensuring a balanced distribution of talent bound by a salary cap in a fantasy sports league

Group AM

Eduardo Mendes, 20240850

Helena Duarte, 20240530

João Freire, 20240528

Mariana Sousa, 20240516

Spring Semester 2024-2025

https://github.com/Dumendes10/CIFO_24_25_Group_AM

Contents

1. Introduction

Fantasy sports leagues have become widely popular among fans, offering them the excitement of transitioning from armchair coaches to active team managers. Participants manage their own teams while competing against each other using real-world players. Success in these leagues depends not only on selecting the best players but also on constructing teams within strict constraints, such as salary caps, position requirements, and player exclusivity.

This project aims to develop a Genetic Algorithm (GA) capable of creating balanced leagues with given players and constraints. Our approach encodes team configurations as individuals in a population and evolves them through selection algorithms, crossover, and mutation operators. We focus on identifying effective strategies for creating fair and competitive leagues by minimizing the difference in teams' average skill levels.

2. Problem Definition

As mentioned above, this project implements a Genetic Algorithm that assigns players to teams in a fantasy sports league. The goal of the optimization problem is to ensure a balanced distribution of talent while staying within a maximum budget of 750M€ per team. Players are characterized by three attributes: skill rating, salary, and role in the team, which can be Goalkeeper (GK), Defender (DEF), Midfielder (MID), or Forward (FWD).

A solution (individual) is a complete league composed of five valid teams, each consisting of 1 GK, 2 DEF, 2 MID, and 2 FWD. The representation is a list with five sub-lists, one per team, always ordered as [GK, DEF, DEF, MID, MID, FWD, FWD]. Players are stored by unique ID, through which their attributes can be accessed on a dataframe of their characteristics. Lists were chosen because they are easy to manipulate. Structuring them this way implicitly preserves role information, simplifying the application of genetic operators.

The search space includes all league configurations where the teams follow the required structure, all players are assigned, and no player is in two teams at once. Configurations violating these rules are excluded. However, leagues whose teams exceed the budget are allowed in the search space, as they may be close to optimal solutions that could become unreachable if excluded. Instead, these configurations are heavily penalized in the fitness function, ensuring they are never selected.

The fitness function evaluates how well balanced a league is. In other words, it sees if the average skill rating of the players is roughly the same among teams. Apart from that, it also enforces the salary cap constraint. To capture this, we compute the standard deviation of the average skill ratings of the five teams. The lower the standard deviation, the more balanced the league, and therefore the better the solution. This knapsack-like problem is, therefore, a minimization challenge. Salary constraints are handled by a penalty mechanism: if any team's total salary exceeds the 750M€ budget, the solution is penalized with a large fitness value (10^9). For a more formal definition of fitness, let s_i be the average skill of team i, for i = 1, …, 5, and let *std(s)* be the standard deviation of s_1, …, s_5. Then the fitness is:

$$fitness = \begin{cases} std(s), & if\ all\ team\ budgets\ in\ the\ league\ \leq\ 750M€ \\ 10^9, & otherwise \end{cases}$$

3. Selection Mechanisms and Genetic Operators

Genetic Algorithms search for optimal results in a space of possible solutions, beginning their search with randomly generated solutions and trying to improve their quality by employing genetic operators. For our genetic operators, we developed three mutation and two crossover mechanisms.

3.1. Mutation Operators

Mutation creates a new individual by modifying a small portion of the syntactic structure of an existing individual. We implemented three mutation operators: *Player Swap*, *Role Shuffle*, and *Player Role Left Shift*.

Player Swap simply switches one player with another in the same position across two teams. This is a minimal, low-risk mutation that preserves the team structure and introduces small variations without much disruption.

Role Shuffle selects a role, gathers all players in that role across the league, shuffles them, and reassigns them to the same positional slots in their teams. This introduces controlled randomness while preserving role constraints, helping the algorithm break free from local optima.

Player Role Left Shift selects a role and rotates the players in that role across teams to the left by a random number of positions. This operator also promotes structured diversity in the solutions while preserving player roles.

Visualizations of these mechanisms can be found in **Appendix 3.1**.

3.2. Crossover Operators

Crossover combines two parent solutions to generate offspring, allowing useful traits from the parents to be inherited and recombined. We implemented two crossover operators: *Standard Crossover with Position Repair*, and *Crossover by Position*.

The *Standard Crossover with Position Repair* performs crossover at the team level: a crossover point is randomly chosen, and the first part of one parent is combined with the second part of the other to produce two offspring. However, standard crossover may result in duplicate players across teams, or players assigned to the wrong positional slot. To ensure that the offspring remain valid solutions, we apply a repair mechanism after crossover. The repair process detects duplicate players, identifies missing players, and replaces duplicates with missing players of the correct role (but randomly if there are more than 1 duplicate for the same position), based on a fixed team structure [GK, DEF, DEF, MID, MID, FWD, FWD].

The *Crossover by Position* performs crossover based on player roles, rather than team structure. Instead of cutting the individual at a fixed point, this method selectively keeps certain positions from one parent and fills the remaining positions from the other. The steps are:

(i) For each team, keep players from Parent 1 in selected roles (e.g., GK and MID);
(ii) Fill the remaining roles from Parent 2, avoiding duplicates and ensuring correct positions; and
(iii) If any positions remain unfilled, complete them using available players of the correct role.

This operator offers more control over genetic inheritance, allowing role-specific traits to be preserved while maintaining global feasibility.

Visualizations for the crossover mechanisms can be found in **Appendix 3.2**.

3.3. Selection Mechanisms

These genetic operators are not applied to all the individuals in the population, only to some that have been selected. Therefore, we developed two selection mechanisms: *Ranking Selection* and *Tournament Selection*. These were chosen because, in class, fitness proportionate selection was implemented, and we wanted to independently develop alternative strategies.

Ranking Selection assigns selection probabilities based on the relative rank of individuals in the population, not their absolute fitness. The steps are:

(i) Sort the population by fitness, from worst to best;
(ii) Assign ranks from 1 to n;
(iii) Compute linear probabilities based on rank; and
(iv) Select one individual via weighted random sampling.

The fact that this method is not sensitive to differences in fitness prevents fitness outliers from dominating the problem. See **Appendix 3.3** for a visualization.

Tournament Selection selects the best individual based on competition within a randomly sampled subset of the population. This method allows us to easily tune the selection pressure by simply changing *k*, the tournament size. It works by:

(i) Sampling, with replacement, a random number of individuals *k*; and
(ii) Selecting the best individual (lowest fitness) out of the tournament group.

We tested different values of k from 2 to 7, and chose the one with best fitness (k=4) **(Appendix 3.4).** Additionally, we designed our method to allow both minimization and maximization by adjusting the sort direction, making it re-useable in the future. A visualization can be found in **Appendix 3.5**.

4. Performance Analysis

To optimize our GA, we tested multiple different configurations, adjusting the hyperparameters to improve performance. Using grid search, we evaluated five combinations of probabilities: two crossover probabilities (xo_prob = 0.25 and 0.44) and three mutation probabilities (mut_prob = 0.29, 0.37, and 0.81). Each combination was tested over 30 iterations, always keeping a fixed population size of 50 individuals and 100 generations. For each run, we recorded the final fitness value.

To check if the differences between combinations were statistically meaningful, we did a Kruskal-Wallis H-test, which confirmed that at least one pair of combinations differed significantly ($H = 23.476$, $p = 0.0001$). We also performed pairwise Mann-Whitney U tests, having found that a higher mutation probability (0.81) consistently outperformed the lower mutation probabilities (0.29 and 0.37), independently of the crossover probability used. For instance, the setup with a xo_prob = 0.25 and

mut_prob = 0.81 achieved the best average fitness score (0.1317), and this difference was statistically significant ($p < 0.01$).

Changing the crossover probability from 0.25 to 0.44 didn't make a significant difference in performance, which suggests that mutation rate plays a much bigger role in how well the genetic algorithm performs.

We also performed Levene's tests to check if the variability ("consistency") of the fitness results was different across the combinations. Even if one configuration had a better average fitness, we wanted to make sure that performance was consistent across iterations. The tests showed no significant differences in variance between the groups, so all configurations had similar stability in their results. Although the best-performing combination (with mut_prob = 0.81) had the lowest standard deviation, this difference wasn't statistically significant, so we can conclude that the GA performance was consistently stable across all tested combinations.

We conclude the mutation operator was the main reason for the differences in the fitness scores (since all the other parameters were the same across iterations), with higher mutation rates helping the algorithm converge to the global solution. Changes in the crossover probability had little impact on the fitness scores. We also conclude the combination with crossover probability of 0.25 and mutation probability of 0.81 achieved statistically the best average fitness (the lower one) but also had the lowest standard deviation. We conclude it's the most effective and reliable, and the best for our genetic algorithm.

To further explore these dynamics and validate our findings, we conducted a total of 12 tests. Each test represented a different parameter configuration, and success was defined as the ability of the GA to consistently build balanced teams in the league, minimizing differences in average skill levels (minimizing standard deviation).

The convergence behavior of our GA is strongly affected by the crossover and mutation operators. As we can see in Appendix 4.1 – Crossover and Mutation Comparisons, different crossovers demonstrate varying rates of fitness improvement. The standard crossover with position repair achieves a lower fitness value faster, which indicates that this strategy is more effective at exploring the solution space.

Similarly, mutation operators' selection has a significant impact. The player swap mutation operator leads to a faster, and more stable convergence to low fitness values. However, the player role left shift operator and role shuffle mutation result in slower convergence, with a higher final fitness. Concluding that the player swap mutation is more effective at introducing beneficial diversity without disrupting good solutions.

Elitism was implemented in our GA. Appendix 4.2 – Mean and Median Fitness Across Generations demonstrates that using elitism consistently results in lower mean fitness values and a narrower confidence interval, indicating a better and more consistent convergence.

The approach that we chose, particularly with the combination of standard crossover with position repair and player swap mutation, achieved good results. The GA was able to minimize the fitness function effectively, with low and stable values. Both the **Appendix 4.2** and **Appendix 4.3** show clear downward trends, with the best performance converging quickly.

For possible improvements, we could expand the grid search to include selection mechanisms, run experiments over more generations, and test larger population sizes. These changes could have helped improve the model's behaviour. However, due to computational power limitations, it was not possible to run more iterations.

5. Conclusion

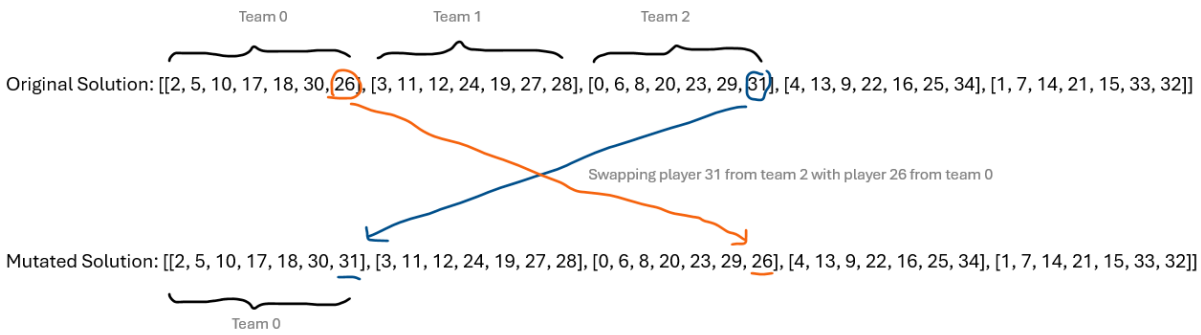
In this project, we managed to build a Genetic Algorithm that effectively creates balanced fantasy leagues under the required constraints. We tested different combinations of crossovers, mutations, and selection methods that we developed along the way, and found that mutation had the biggest impact on performance, while elitism didn't have such an impact in our model.

Although we achieved good results, we didn't include selection methods in the grid search or explore larger populations due to limited time and computational resources, which would be clear next steps for improvement. We also observed some signs of premature convergence in configurations with limited diversity, which could be addressed in future work. Still, the algorithm performed well and helped us successfully optimize the problem we set out to solve.

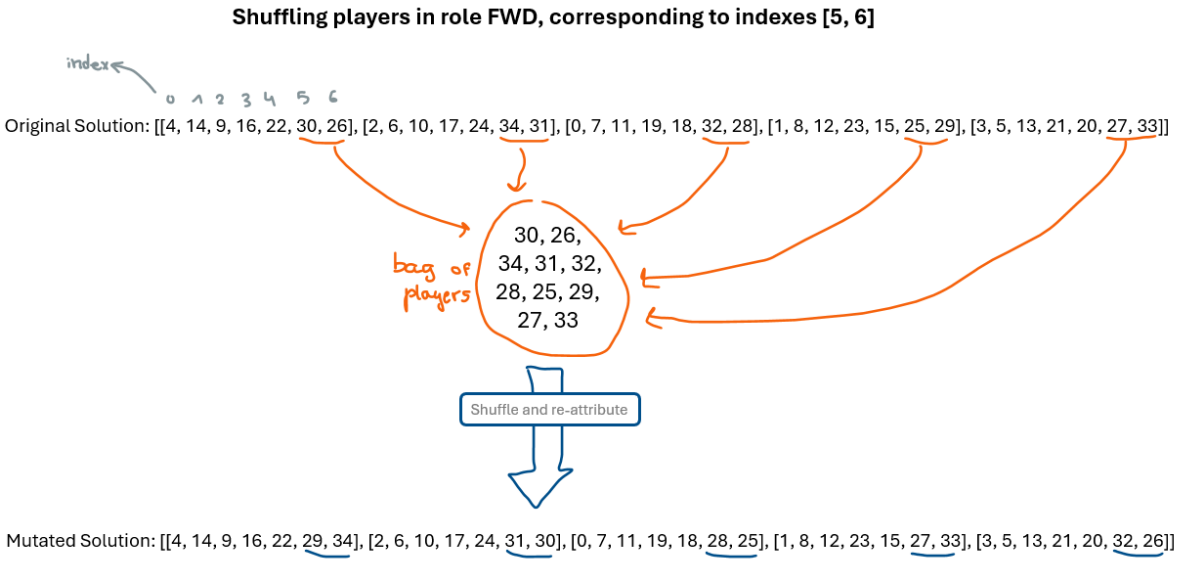
6. Appendix

Appendix 3.1 – Mutation operators

Player Swap Mutation:

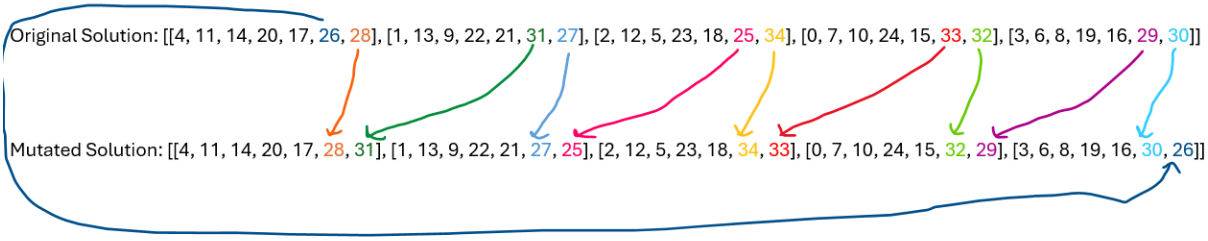


Role Shuffle Mutation:



Player Role Left Shift Mutation:

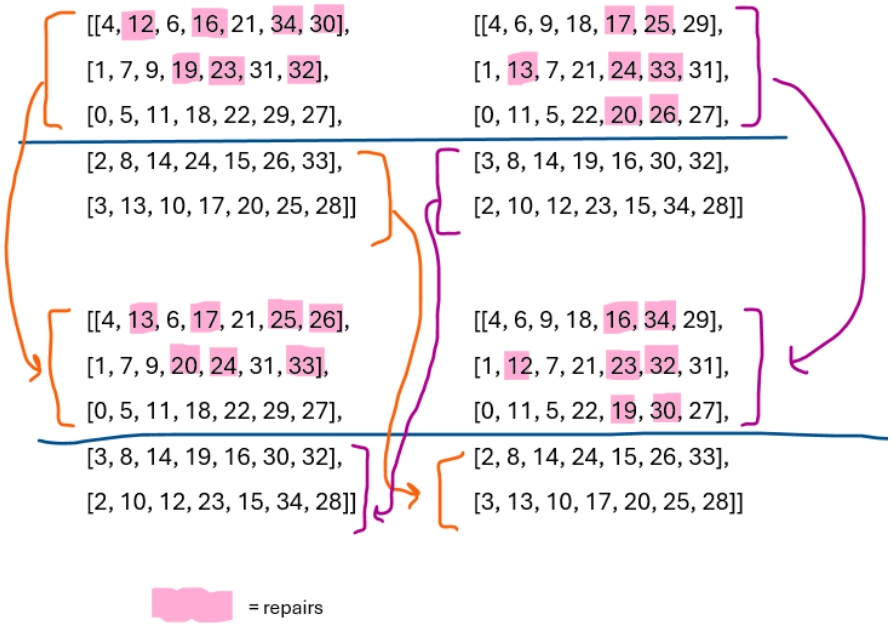
Shifting role group FWD, corresponding to indexes [5, 6], by 1 position



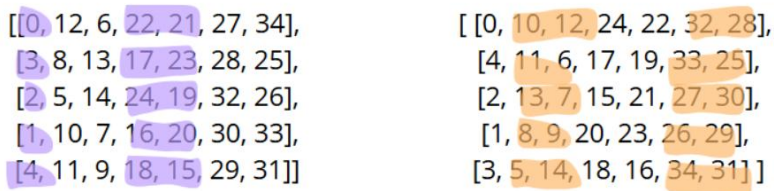
Original position	Team 0, index 5	Team 0, index 6	Team 1, index 5	Team 1, index 6	Team 2, index 5	Team 2, index 6	Team 3, index 5	Team 3, index 6	Team 4, index 5	Team 4, index 6
Final position	Team 4, index 6	Team 0, index 5	Team 0, index 6	Team 1, index 5	Team 1, index 6	Team 2, index 5	Team 2, index 6	Team 3, index 5	Team 3, index 6	Team 4, index 5

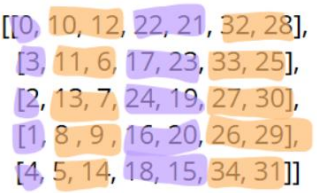
Appendix 3.2 – Crossover operators

Standard crossover with position repair:



Crossover by position:





Appendix 3.3 – Rank selection

Individual	Fitness
i_1	3
i_2	4
i_3	2

$$P(\text{selecting } i) = \frac{rank_i}{\sum_{j=1}^{N} rank_j}$$

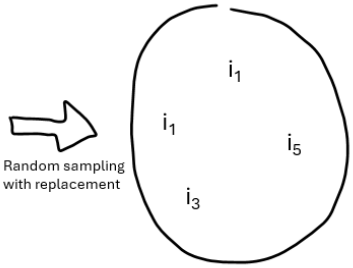
Rank	Individual	Probability
1	i_3	1/6
2	i_1	2/6
3	i_2	3/6

Appendix 3.4 – Tournament selection, Manual Search

k	Population Size	Generations	Best Fitness
4	50	200	0.057
4	50	100	0.057
2	50	100	0.100
3	50	100	0.110
5	50	100	0.139
6	50	100	0.106
7	50	100	0.140

Appendix 3.5 – Tournament selection

Individual	Fitness
i_1	3
i_2	4
i_3	2
i_4	10
i_5	7
i_6	1
i_7	8
i_8	5

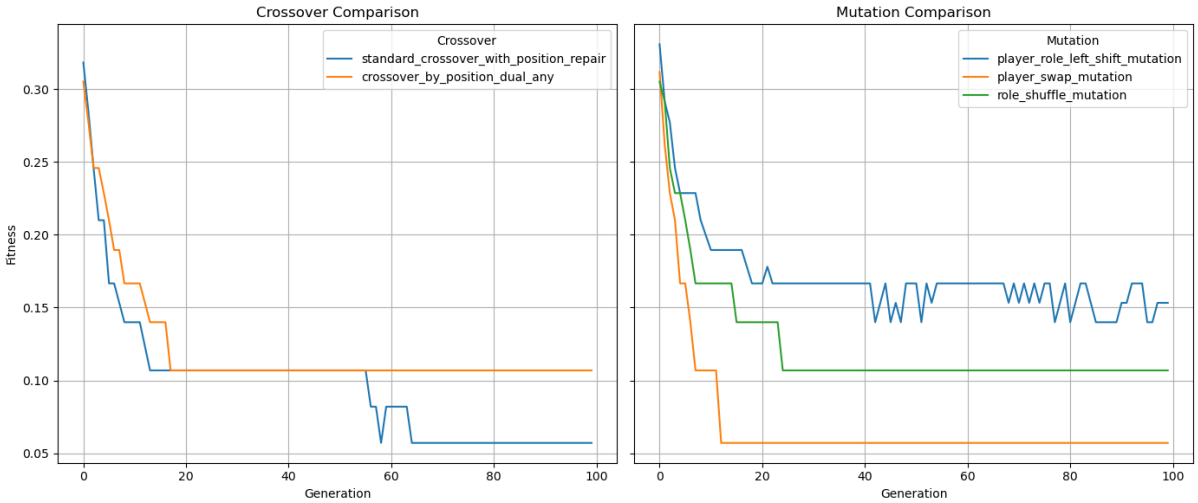


Order by decreasing fitness (minimization problem)

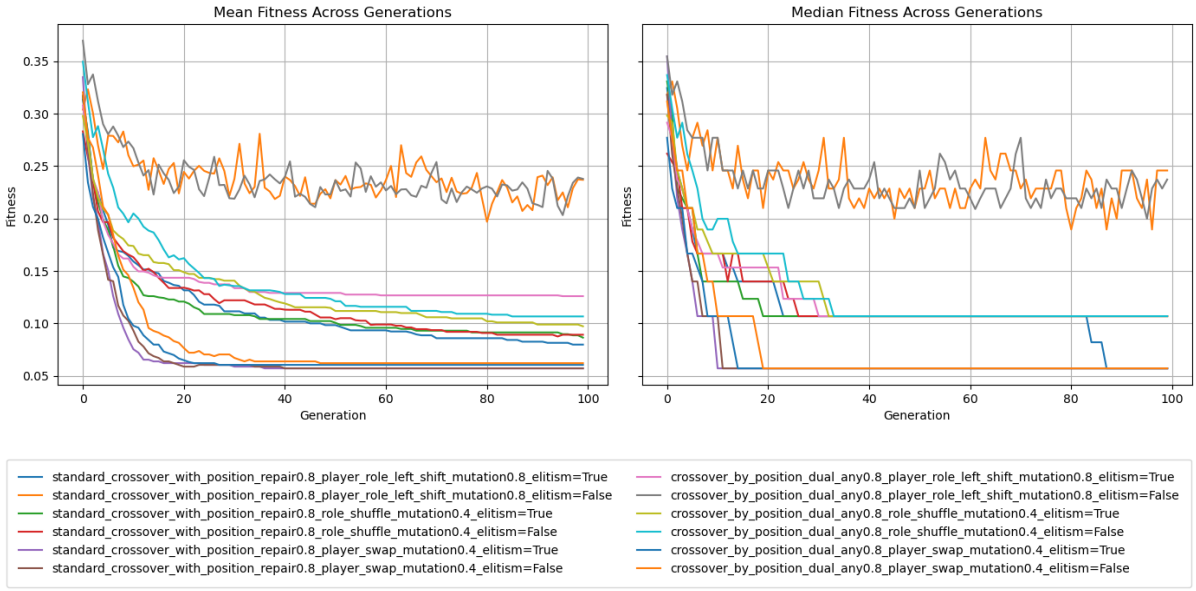
Winner

Individual	Fitness
i_3	2
i_1	3
i_1	3
i_5	7

Appendix 4.1 – Crossover and Mutation Comparisons



Appendix 4.2 – Mean and Median Fitness Across Generations



Appendix 4.3 – Mean and Median Fitness by Crossover and Mutation Operator