Fantasy Football Optimization

Marco De Rito - SM3800016

University of Trieste

Trieste, Italy

marco.derito@studenti.units.it

Abstract

This report represents my final project for an advanced optimization course, where I develop a comprehensive methodology for fantasy football team selection, formulated as an *inverse optimization problem*. I apply three well-known meta-heuristics: Particle Swarm Optimization (PSO), Differential Evolution (DE) and Evolution Strategies (ES) in a multi-manager auction setting, ensuring budget and positional constraints are respected.

The study integrates theoretical foundations (including PSO, DE, ES, and inverse optimization principles), practical code implementations in Python (data loading, scoring, fitness function, forced assignments, multi-round auctions), and empirical evaluations. Through comparisons of budget usage, forced picks, and final team scores, I show that each algorithm delivers competitive results, albeit with differences in strategy diversity and resource utilization. Finally, I reflect on future enhancements.

CCS Concepts

• **Computing methodologies → Optimization algorithms**.

Keywords

Fantasy Football, Inverse Optimization, Particle Swarm Optimization, Differential Evolution, Evolution Strategies, Machine Learning, Auction Mechanisms, Student Project

1 Introduction

Fantasy football is a popular game where participants (managers) build squads of real-life players under specific rules (budget limits, positional minimums, etc.). In this project, I aim to formulate this selection task as an *inverse optimization* scenario: rather than defining player values arbitrarily, I adapt a scoring function based on real historical stats (goals, assists, ratings, etc.).

I employ three established methods from the field of *evolutionary computation* and *swarm intelligence*:

- **Particle Swarm Optimization (PSO)**
- **Differential Evolution (DE)**
- **Evolution Strategies (ES)**

Each manager picks one algorithm to generate bids for available players in a multi-manager auction. After repeated rounds, I resolve conflicts and enforce forced assignments if rosters remain incomplete. This report summarizes my approach, from conceptual underpinnings to Python implementation details, culminating in comparative charts of each algorithm's performance.

1.1 Report Structure

- **Section 2** introduces the concept of inverse optimization for fantasy football and briefly reviews the fundamentals of PSO, DE, and ES.
- **Section 3** describes the project's architecture (data loading, utility classes, the multi-manager auction routine) and provides code snippets for the scoring function, common fitness logic, and forced assignments.
- **Section 4** presents empirical analyses: budget usage, forced picks, and average team scores.
- **Section 5** concludes with reflections and potential future extensions, such as hybrid evolutionary approaches or reinforcement learning.

2 Theoretical Background

2.1 Inverse Optimization in Fantasy Football

An *inverse optimization* framework attempts to refine a cost or reward function so that historically observed solutions appear near-optimal. In fantasy football, the logic is as follows: if certain players consistently yield high performance, I want my scoring function to reflect that. While I do not explicitly solve a full parameter-fitting routine, I adopt the spirit of inverse optimization by letting real data guide the final scoring weights (`score_player`).

2.2 Particle Swarm Optimization (PSO)

PSO is inspired by natural swarms. I track positions $\mathbf{x}_i$ (bids) and velocities $\mathbf{v}_i$ for each particle i, updated as:

$$\mathbf{v}_i(t+1) = \omega \mathbf{v}_i(t) + c_1 r_1 (\mathbf{p}_i - \mathbf{x}_i(t)) + c_2 r_2 (\mathbf{g} - \mathbf{x}_i(t)),$$
$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1).$$

Where:

- $\mathbf{x}_i(t)$ – position of particle i at iteration t
- $\mathbf{v}_i(t)$ – velocity of particle i at iteration t
- $\mathbf{p}_i$ – personal best position found by particle i
- $\mathbf{g}$ – global best position in the swarm
- r_1, r_2 – random numbers $\in [0, 1]$
- c_1, c_2 – cognitive and social acceleration coefficients
- ω – inertia weight (controls momentum)

$\mathbf{p}_i$ is the best solution found by particle i, and $\mathbf{g}$ is the global best. PSO often rapidly explores a vast solution space but can exhibit variability if not properly tuned.

2.3 Differential Evolution (DE)

DE uses vector differences to generate mutant solutions. For each agent (bid vector). The mutation and crossover steps in DE are defined as:

$$\mathbf{y} = \mathbf{x}_a + F \cdot (\mathbf{x}_b - \mathbf{x}_c)$$

$$\mathbf{z}_j = \begin{cases} \mathbf{y}_j & \text{if } r_j < CR \text{ or } j = j_{rand} \\ \mathbf{x}_{i,j} & \text{otherwise} \end{cases}$$

Where:

- $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ – distinct randomly chosen vectors ($\neq \mathbf{x}_i$)
- F – mutation factor, here drawn uniformly from the range [0.5, 1.0]
- CR - crossover rate, here: 0.7
- $\mathbf{y}$ – mutant vector
- $\mathbf{z}$ – trial vector
- j_{rand} – random index to ensure at least one component from $\mathbf{y}$

Then it crosses over $\mathbf{y}$ with the target $\mathbf{x}_i$. If the resulting trial $\mathbf{z}$ has better fitness, it replaces $\mathbf{x}_i$. DE is known for robust fine-tuning in continuous spaces.

2.4 Evolution Strategies (ES)

ES employ Gaussian mutations plus a survivor selection scheme $(\mu + \lambda)$ or (μ, λ). By mutating each parent to create offspring, then retaining the top solutions, ES preserve high diversity. In the fantasy scenario, random mutation of bids can sometimes yield unexpected but valuable team compositions, especially if guided by a well-structured penalty for budget or positional violations.

In Evolution Strategies, offspring are generated via Gaussian mutation and then selected according to the $(\mu + \lambda)$ strategy. The mutation is defined as:

$$\mathbf{x}_{\text{child}} = \mathbf{x}_{\text{parent}} + \sigma \cdot \mathcal{N}(0, I)$$

Where:

- $\mu = 40$ – number of parents
- $\lambda = 80$ – number of offspring
- σ – mutation strength (implicit via DEAP's toolbox)
- $\mathcal{N}(0, I)$ – standard Gaussian noise
- Selection is $(\mu + \lambda)$: best μ individuals from parents and offspring survive

3 Implementation and Architecture

3.1 Overall Project Structure

My Python project is split into modules:

- `data_loader.py` - reading and cleaning the CSV with player stats
- `utils.py` - classes `Player`, `Manager`, scoring function
- `optimization.py` - all the meta-heuristics (PSO, DE, ES) plus the multi-manager auction logic
- `main.py` - orchestrates user inputs, runs the auction, generates final summary
- `report_generator.py` - builds a PDF with charts (budget usage, forced picks, etc.)

3.2 Scoring Function and Common Fitness Logic

One of the most critical elements is the player scoring function, which turns historical stats (goals, assists, etc.) into a single numeric value. Compute the player's score based on various performance metrics.

Scoring breakdown:

- Goals Scored: +0.5 per goal
- Assists: +0.2 per assist
- Yellow Cards: -0.05 per card
- Red Cards: -0.1 per card
- Rating: +0.2 per rating
- Penalties Scored: +0.2 per penalty
- Goals Conceded: -0.5 per goal
- Penalties Saved: +0.5 per save
- Matches Played: +0.5 per match

Parameters:

- player: The Player object for which to calculate the score.

Returns:

- A numerical value representing the player's overall score.

```python
def score_player(player):

    # Retrieve player statistics using getattr to
        provide default values if not set
    goals = getattr(player, 'goals_scored', 0)
    assists = getattr(player, 'assists', 0)
    yellow_cards = getattr(player, 'yellow_cards', 0)
    red_cards = getattr(player, 'red_cards', 0)
    rating = getattr(player, 'rating', 6.0)
    penalties = getattr(player, 'penalties_scored',
        0)
    goals_conceded = getattr(player, 'goals_conceded'
        , 0)
    penalties_saved = getattr(player, '
        penalties_saved', 0)
    matches_played = getattr(player, 'matches_played'
        , 0)

    # Calculate the score based on the weighted
        metrics
    score = ((0.5 * goals) + (0.2 * assists) - (0.05
        * yellow_cards) - (0.1 * red_cards) + (0.2 *
         rating) + (0.2 * penalties)) - (0.5 *
        goals_conceded) + (0.5 * penalties_saved) +
        (0.5 * matches_played)
    return score
```

Listing 1: Example scoring function in utils.py

Each algorithm then needs a **fitness function** to evaluate how good a bid vector is. If an algorithm is a minimizer (like `pyswarm.pso` or `differential_evolution`), I return negative total score (plus large penalties for invalid solutions). The snippet below shows a common approach (`common_fitness_logic`):

```python
def common_fitness_logic(manager, bids: List[float],
    roles: List[str], scores: List[float], min_thr: int)
    -> float:

    global evaluation_count, surrogate_model
    evaluation_count += 1

    int_bids = [int(round(b)) for b in bids]
    for i, bid_value in enumerate(int_bids):
            if 0 < bid_value < min_thr:
                    int_bids[i] = min_thr

    budget = manager.budget
    total_spent = sum(int_bids)
    if total_spent > budget:
            return HIGH_PENALTY

    for bid_value in int_bids:
            if bid_value > budget *
                SINGLE_PLAYER_CAP_RATIO:
                    return HIGH_PENALTY

    leftover_budget = budget - total_spent
```

```
21          max_total = int(manager.max_total)
22          players_needed_local = max_total - len(manager.
                team)
23          if leftover_budget < players_needed_local:
24                  return HIGH_PENALTY
25
26          leftover_penalty = ((leftover_budget -
                players_needed_local) ** BUDGET_LEFTOVER_EXP
                ) * LEFTOVER_MULTIPLIER
27          chosen_count = sum(1 for v in int_bids if v >=
                min_thr)
28          penalty = abs(chosen_count - players_needed_local
                ) * PLAYER_COUNT_PENALTY
29
30          role_count = {}
31          for i, bid_value in enumerate(int_bids):
32                  if bid_value >= min_thr:
33                          r = roles[i]
34                          role_count[r] = role_count.get(r,
                              0) + 1
35
36          for r, (min_r, max_r) in manager.role_constraints
                .items():
37                  current_have = sum(1 for p in manager.
                      team if p.role == r)
38                  add_count = role_count.get(r, 0)
39                  if current_have + add_count < min_r or
                      current_have + add_count > max_r:
40                          return HIGH_PENALTY
41
42          penalty += leftover_penalty
43
44          total_score = 0.0
45          for i, bid_value in enumerate(int_bids):
46                  if bid_value >= min_thr:
47                          w = role_weight(manager, roles[i
                              ])
48                          total_score += w * scores[i]
49
50          computed_fitness = penalty - total_score
51
52          if USE_SURROGATE and surrogate_model is not None:
53                  surrogate_model.update(bids,
                      computed_fitness)
54                  if evaluation_count % 20 == 0:
55                          surrogate_model.train()
56                  if evaluation_count >=
                      SURROGATE_THRESHOLD:
57                          surrogate_val = surrogate_model.
                              evaluate(bids)
58                          return 0.5 * computed_fitness +
                              0.5 * surrogate_val
59
60          return computed_fitness
```

Listing 2: Common fitness logic for PSO/DE/ES

3.3 Manager Strategies: PSO, DE, ES

In my code, each `Manager` calls a function `decide_bids` that internally picks the appropriate meta-heuristic.

The performance of evolutionary algorithms is highly dependent on the appropriate selection of their parameters. Below is a summary of the key parameters for PSO, DE, and ES, along with their descriptions, typical values, and justifications.

Table 1: Key Parameters for PSO, DE, and ES

Alg.	Parameter	Description	Typical Values
PSO	ω (Inertia Weight)	Balances global and local exploration.	0.7–0.9; higher values promote global search.
PSO	c_1 (Cognitive Coefficient)	Influence of personal best position.	1.5–2.0; standard choice to balance exploration and exploitation.
PSO	c_2 (Social Coefficient)	Influence of global best position.	1.5–2.0; ensures convergence toward the global best.
DE	F (Differential Weight)	Scales the differential variation.	0.5–0.8; controls amplification of difference vectors.
DE	CR (Crossover Rate)	Probability of parameter crossover.	0.7–0.9; higher values increase diversity.
ES	μ (Parent Population Size)	Number of parent solutions.	Typically $\lambda/7$; balances selection pressure.
ES	λ (Offspring Population Size)	Number of offspring generated.	Typically 7μ; ensures sufficient exploration.

The chosen values for these parameters are based on standard practices in evolutionary computation. For instance, in PSO, an inertia weight (ω) around 0.7–0.9 is commonly used to balance exploration and exploitation. Similarly, in DE, a crossover rate (CR) of 0.7–0.9 is typical to maintain diversity among solutions. In ES, setting λ approximately seven times μ is recommended to maintain an appropriate selection pressure.

```
1       from pyswarm import pso
2
3       best_bids, _ = pso(
4               fitness_func, lb, ub,
5               swarmsize=40,
6               maxiter=80,
7               omega=0.7,
8               phip=1.8,
9               phig=1.8
10      )
```

Listing 3: PSO parameter setup with pyswarm

For PSO, I set the inertia weight $\omega = 0.7$, and both cognitive and social coefficients $c_1 = c_2 = 1.8$. These values were chosen to balance exploration and exploitation, ensuring stable convergence while allowing the swarm to effectively explore the solution space.

```
1       from scipy.optimize import differential_evolution
2
3       result = differential_evolution(
4       fitness_wrapper,
5       bounds=[(0.0, float(max_bid_per_player))] * n,
6       strategy='best1bin',
7       mutation=(0.5, 1.0),        # F  ~ U(0.5, 1.0)
8       recombination=0.7,          # CR
9       maxiter=50,
10      popsize=15
11      )
```

Listing 4: DE parameter setup with SciPy

These values were chosen to maintain balance between broad mutation and reliable recombination, using SciPy's standard strategy `best1bin`.

```
1       from deap import algorithms
2
3       algorithms.eaMuPlusLambda(
```

```
 4              population_, toolbox,
 5              mu=40,
 6              lambda_=80,
 7              cxpb=0.5,
 8              mutpb=0.3,
 9              ngen=40,
10              verbose=False
11          )
```

Listing 5: ES setup with DEAP using (μ+λ)

I adopted the ($\mu + \lambda$) strategy with $\mu = 40$ and $\lambda = 80$ to ensure diversity among offspring. The mutation probability was set to 0.3, and crossover to 0.5, as commonly suggested for continuous optimization.

3.4 Multi-manager Auction and Forced Assignments

Once each manager decides their bids in a given round, I collect them and assign players to the highest bidder or use a small function `resolve_competition` to handle tie-breaks. Below is a simplified version of my multi-round auction orchestrator:

```
 1
 2      def multi_manager_auction(players, managers,
            max_turns=30):
 3      not_assigned = {p.pid: p for p in players}
 4      turn = 0
 5
 6      while turn < max_turns and not_assigned:
 7          turn += 1
 8          all_bids = []
 9
10      # Each manager proposes bids
11      for mgr in managers:
12              unass_list = list(not_assigned.values())
13      bids = mgr.decide_bids(unass_list)
14      for (pid, amt) in bids:
15              if pid in not_assigned and mgr.can_buy(
                    not_assigned[pid], amt):
16              all_bids.append((mgr, pid, amt))
17
18      if not all_bids:
19              break
20
21      # Group bids by player
22      bids_by_player = {}
23      for (mgr, pid, amt) in all_bids:
24              bids_by_player.setdefault(pid, []).append
                    ((mgr, amt))
25
26      # Resolve conflicts and assign
27      for pid, manager_offers in bids_by_player.items()
            :
28              if pid not in not_assigned:
29                      continue
30
31      if len(manager_offers) == 1:
32              best_manager, best_amt = manager_offers
                    [0]
33      else:
34              best_manager, best_amt =
                    resolve_competition(manager_offers)
35
36      # Assign if feasible
37      if best_manager.can_buy(not_assigned[pid],
            best_amt):
38              player_obj = not_assigned[pid]
39              player_obj.assigned_to = best_manager.
                    name
40              player_obj.final_price = best_amt
41              best_manager.update_roster(player_obj,
                    best_amt)
42              del not_assigned[pid]
```

```
43
44      # Post-process forced assignments
45      forced_assignments(managers, list(not_assigned.
            values()))
46      return managers, list(not_assigned.values())
```

Listing 6: Multi-manager auction framework

Finally, the `forced_assignments` step ensures each manager meets the *minimum role constraints* by forcibly adding leftover players (often at base cost 1). This is a fallback mechanism for any manager who runs out of budget or gets outbid in key positions:

```
 1 def forced_assignments(managers, leftover_players):
 2      for mgr in managers:
 3      for role, (min_r, _) in mgr.role_constraints.
            items():
 4      # Count how many players of this role the manager
            already has
 5      count_current = sum(p.role == role for p in mgr.
            team)
 6
 7      # Keep assigning until the minimum requirement is
            met
 8      while count_current < min_r:
 9              # Pick the best remaining player of the
                    needed role
10              try:
11              candidate = next(
12              p for p in sorted(
13              leftover_players,
14              key=score_player,
15              reverse=True
16              )
17              if p.role == role
18              )
19              except StopIteration:
20              # No more players of this role are
                    available
21              break
22
23              # Abort if the manager has no budget left
24              if not mgr.can_buy(candidate, 1):
25              break
26
27              # Apply the forced assignment at the
                    minimum price (1 credit)
28              candidate.assigned_to = mgr.name
29              candidate.final_price = 1
30              mgr.update_roster(candidate, 1)
31
32              # Remove the player from the global pool
33              leftover_players.remove(candidate)
34
35              # Update the current count for this role
36              count_current += 1
```

Listing 7: Simple forced assignment to fill leftover roles

4 Example: Initial Inputs and Scoring Calculation

In this section, I present an overview of the initial data and results used in my optimization process. I include several graphical and tabular analyses to illustrate the diversity of input data and the effectiveness of my algorithm in handling budget constraints and role requirements. The following figures and tables were generated during simulation runs, and their analysis supports the conclusion that my approach is both effective and robust.

4.1 Graphical Analyses

4.1.1 Manager Distribution. Figure 1 shows the distribution of managers by strategy (PSO, DE, and ES). This graph confirms that my experiment has a balanced input—each strategy is well-represented.
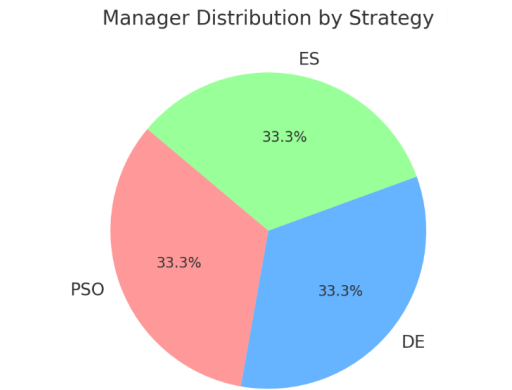

Figure 1: Distribution of managers by strategy.

4.1.2 Player Score Distribution. Figure 2 displays the distribution of player scores based on historical performance metrics. Notice that most players score in the lower-to-mid range, with only a few high-scoring outliers. This distribution forces managers to carefully allocate their budget. The effectiveness of my algorithm is demonstrated by its ability to select the optimal balance between expensive top performers and numerous affordable players.
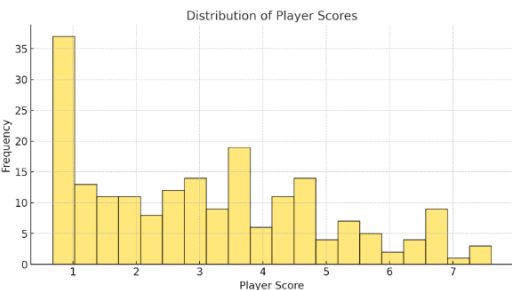

Figure 2: Distribution of player scores based on historical performance.

4.1.3 Budget Usage. Figure 3 illustrates the budget utilization across managers. The graph shows that most managers spend nearly all of their available budget, which is an indication that the algorithm efficiently utilizes resource. Effective budget usage is critical in ensuring that no valuable credits remain unspent, thus maximizing the potential overall team score.
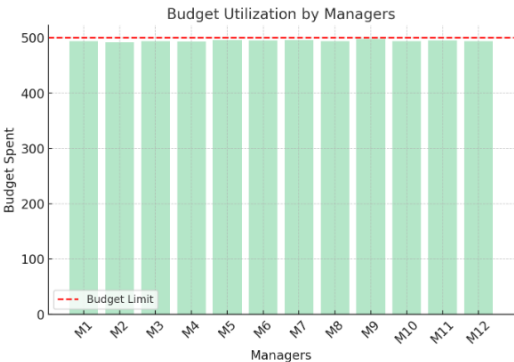

Figure 3: Budget Usage by Managers.

4.1.4 Forced Assignments. Figure 4 depicts the number of forced assignments per manager. Forced assignments occur when a manager fails to naturally satisfy the minimum role requirements, prompting the algorithm to assign additional players at a base cost. A low number of forced assignments indicates that the optimization process is generally successful at forming complete teams without needing extra interventions, thus proving the robustness of the method.
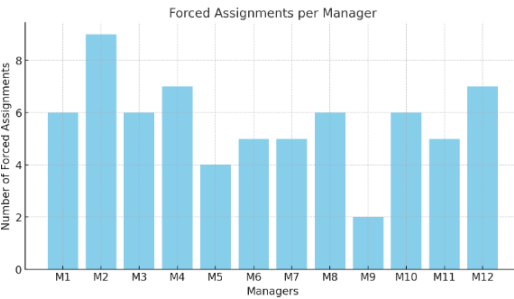

Figure 4: Number of Forced Assignments per Manager.

4.2 Tabular Analyses

4.2.1 Manager Recap Table. Table 2 provides a detailed recap of each manager's performance. It includes the chosen strategy, the number of forced assignments, the total budget spent, and the final objective score (team score). A low number of forced assignments along with near 500 spent budget demonstrates that the algorithm effectively satisfies all constraints.

Table 2: Recap Table: Manager Stats

Manager	Strat.	Forced	Spent	Tot. Score
Manager_1	PSO	6	494.0	52.91
Manager_2	DE	9	492.0	51.48
Manager_3	PSO	6	494.0	44.46
Manager_4	DE	7	493.0	53.64
Manager_5	ES	4	496.0	49.18
Manager_6	ES	5	495.0	46.02
Manager_7	PSO	5	496.0	47.00
Manager_8	DE	6	494.0	55.32
Manager_9	ES	2	498.0	44.23
Manager_10	DE	6	494.0	46.59
Manager_11	ES	5	495.0	39.99
Manager_12	PSO	7	494.0	53.23

4.2.2 Performance by Strategy Table. Table 3 aggregates the performance of each strategy, reporting the number of managers using each method along with the average total score. This table confirms that although DE tends to achieve a slightly higher total score, the differences among the strategies are minimal—further evidence of the algorithm's robustness.

Table 3: Performance by Strategy

Strat.	Man.	Avg Tot. Score	Avg Forced	Avg Spent
PSO	4	49.90	6	494.50
DE	4	51.76	7	493.25
ES	4	44.85	4	496.00

While DE yielded the highest average total score, it also required the most forced assignments, indicating a tendency to overconcentrate on top players and budget usage, sometimes at the cost of team completeness. ES, conversely, produced lower scores but needed fewer forced picks, showing a more balanced yet less aggressive optimization behaviour.

4.2.3 Player Score Summary Table. Table 4 summarizes the overall range of player scores calculated by my scoring function. The best, worst, and average scores reflect the challenging nature of the selection process, emphasizing that only a few elite players achieve high scores. This forces managers to balance high-cost, high-score players with more affordable options, demonstrating the effectiveness of my multi-objective optimization.

Table 4: Player Score Summary

	Best	Worst	Average
Player Scores	12.90	0.68	2.84

4.3 Overall Discussion

The integration of these analyses demonstrates that my algorithm is effective and robust:

- The balanced manager distribution (Figure 1) ensures diverse search behaviour.
- The player score distribution (Figure 2) highlights the need for careful budget allocation.
- Budget usage (Figure 3) and forced assignments (Figure 4) indicate that managers nearly fully utilize their budgets while rarely needing forced interventions.
- The high average team scores (Table 3) confirm that my optimization process consistently produces competitive teams.
- The player score summary (Table 4) emphasizes that only a few players reach elite performance levels, reinforcing the need for strategic selection.

Collectively, these results support the conclusion that my multi-manager auction framework effectively balances budget, role constraints, and team performance, validating the effectiveness of my inverse optimization approach in fantasy football.

5 Conclusion and Future Directions

This project combined **evolutionary computation** (PSO, DE, ES) and **inverse optimization** concepts to tackle a multi-manager fantasy football auction. Each strategy effectively balanced budget constraints and role requirements, consistently finding strong teams.

Key Observations:

- **PSO** produced competitive line-ups with relatively stable scores. The average number of forced picks was 6, indicating that PSO was fairly efficient in meeting team constraints, though some forced interventions were necessary.
- **DE** achieved the highest average team score among the three strategies, demonstrating its effectiveness in player selection. However, managers using DE tended to leave small portions of the budget unspent, suggesting potential inefficiencies in budget allocation. This could indicate that the algorithm does not perfectly optimize resource usage. Furthermore, it required the highest number of forced assignments (7), suggesting that aggressive optimization to maximize scores may compromise role coverage and team balance.
- **ES** generated line-ups with the lowest average score but needed the fewest forced picks (4), highlighting its more cautious approach that preserves diversity and respects team role constraints in a more balanced way. It used the budget more fully, with an average spend of 496.00, indicating a less parsimonious use of resources compared to DE, but with lower overall performance.

Future Enhancements:

- **User-Friendly Interface and Web Platform**: Develop a graphical user interface (GUI) or a web-based platform to make the system more accessible and intuitive for managers. This interface could guide users through data input and configuration, and offer visual feedback on optimization progress.
- **Automated Line-up Suggestions**: Extend the system to not only propose a full roster but also suggest an optimal starting line-up based on current form and projected performance, thus assisting managers in making final tactical decisions.

- **Parameter Sensitivity and Auto-Tuning**: Future work could explore testing each meta-heuristic with varying parameters to analyse their impact on convergence, forced assignments, and final score. Furthermore, machine learning techniques (e.g., Bayesian optimization or reinforcement learning) could be applied to automatically tune parameters or even adapt the fitness function dynamically.

Even as a implementation, the results show that *meta-heuristics + domain-specific constraints* can produce realistic, high-quality fantasy rosters, and the interplay of different strategies leads to a fair, competitive multi-manager environment. I hope this framework serves as a foundation for more advanced or hybrid systems in the future.