Fantasy Football Optimization

Marco De Rito - SM3800016
University of Trieste
Trieste, Italy
marco.derito@studenti.units.it

Abstract

This paper 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 metaheuristics: 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 convergence speed and strategy diversity. Finally, I reflect on future enhancements such as hybrid approaches and reinforcement learning, emphasizing how these techniques can further improve automated draft systems in both fantasy and real-world resource-allocation contexts.

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 student 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, we resolve conflicts and enforce forced assignments if rosters remain incomplete. This paper summarizes my approach, from conceptual underpinnings to Python implementation details, culminating in comparative charts of each algorithm's performance.

1.1 Paper 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 ??** 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, we want our 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. We 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 $\mathbf{p}_i$ 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) $\mathbf{x}_i$, it randomly picks three distinct agents $\mathbf{x}_a, \mathbf{x}_b, \mathbf{x}_c$ and forms:

$$\mathbf{y} = \mathbf{x}_a \;\; F\mathbf{x}_b - \mathbf{x}_c.$$

Exam Global and Multi-Objective Optimization, 2025, Trieste, Italy
2025. ACM ISBN none
https://doi.org/none

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 μ λ 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.

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 metaheuristics (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
- Fantasy Rating: +0.2 times the rating
- Penalties Scored: +0.2 per penalty

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, 'fantasy_rating', 6.0)
        penalties = getattr(player, 'penalties_scored', 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))
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

        # Convert budget to float for numerical comparisons
        budget = to_float(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
        max_total = int(to_float(manager.max_total))
        players_needed_local = max_total - len(manager.team)
        if leftover_budget < players_needed_local:
                return HIGH_PENALTY

        leftover_penalty = ((leftover_budget - players_needed_local) ** BUDGET_LEFTOVER_EXP) * LEFTOVER_MULTIPLIER
        chosen_count = sum(1 for v in int_bids if v >= min_thr)
        penalty = abs(chosen_count - players_needed_local) * PLAYER_COUNT_PENALTY

        role_count = {}
        for i, bid_value in enumerate(int_bids):
                if bid_value >= min_thr:
                        r = roles[i]
                        role_count[r] = role_count.get(r, 0) + 1

        for r, (min_r, max_r) in manager.role_constraints.items():
                current_have = sum(1 for p in manager.team if p.role == r)
                add_count = role_count.get(r, 0)
                if current_have + add_count < min_r or current_have + add_count > max_r:
                        return HIGH_PENALTY

        penalty += leftover_penalty
        total_score = 0.0
        for i, bid_value in enumerate(int_bids):
                if bid_value >= min_thr:
                        w = role_weight(manager, roles[i])
                        total_score += w * scores[i]

        computed_fitness = penalty - total_score

        if USE_SURROGATE and surrogate_model is not None:
                surrogate_model.update(bids, computed_fitness)
                if evaluation_count % 20 == 0:
                        surrogate_model.train()
                if evaluation_count >= SURROGATE_THRESHOLD:
                        surrogate_val = surrogate_model.evaluate(bids)
                        return 0.5 * computed_fitness + 0.5 * surrogate_val

        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 metaheuristic. Here is a sketch of the PSO approach, referencing `pyswarm`:

```
from pyswarm import pso
import numpy as np
def manager_strategy_pso(manager, players_not_assigned):
# Convert budget and max_total to single numbers.
        budget = to_float(manager.budget)
        max_total = int(to_float(manager.max_total))
        if budget <= 0 or (max_total - len(manager.team)) <= 0:
                return []
        mb_possible = max_bid_possible(manager)
        max_bid_per_player = to_float(min(max_bid_for_player(manager), mb_possible))
        if max_bid_per_player < 1:
                return []
        n = len(players_not_assigned)
        if n == 0:
                return []
# Create lower bound (lb) and upper bound (ub) as 1D arrays of floats.
        lb = np.array([0.0 for _ in range(n)], dtype=np.float64)
        ub = np.array([max_bid_per_player for _ in range(n)], dtype=np.float64)
        pids = [pl.pid for pl in players_not_assigned]
        roles = [pl.role for pl in players_not_assigned]
        scores = [score_player(pl) for pl in players_not_assigned]
        min_thr = min_bid_threshold(manager)
        def fitness_func(bids_vector: List[float]) -> float:
        return common_fitness_logic(manager, bids_vector, roles, scores, min_thr)
        best_bids, _ = pso(
                fitness_func, lb, ub,
                swarmsize=40,
                maxiter=80,
                omega=0.7,
                phip=1.8,
                phig=1.8
                )
        final_bids = [int(round(b)) for b in best_bids]
        for i, bid_value in enumerate(final_bids):
                if 0 < bid_value < min_thr:
                        final_bids[i] = min_thr
        results = []
        for i, bid_value in enumerate(final_bids):
                if 0 < bid_value <= budget:
                        results.append((pids[i], bid_value))
        return results
```

Listing 3: manager strategy pso

You can similarly define `manager_strategy_de` (using `scipy.optimize.differential_evolution`) and `manager_strategy_es` (using `deap`) by following a similar pattern and calling `common_fitness_logic` internally.

3.4 Multi-manager Auction and Forced Assignments

Once each manager decides their bids in a given round, we 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:

```

        def multi_manager_auction(players, managers, max_turns=30):
        not_assigned = {p.pid: p for p in players}
        turn = 0

        while turn < max_turns and not_assigned:
                turn += 1
                all_bids = []

        # Each manager proposes bids
        for mgr in managers:
                unass_list = list(not_assigned.values())
        bids = mgr.decide_bids(unass_list)
        for (pid, amt) in bids:
                if pid in not_assigned and mgr.can_buy(not_assigned[pid], amt):
                all_bids.append((mgr, pid, amt))

        if not all_bids:
                break

        # Group bids by player
        bids_by_player = {}
        for (mgr, pid, amt) in all_bids:
                bids_by_player.setdefault(pid, []).append((mgr, amt))

        # Resolve conflicts and assign
        for pid, manager_offers in bids_by_player.items():
                if pid not in not_assigned:
                        continue

        if len(manager_offers) == 1:
                best_manager, best_amt = manager_offers[0]
        else:
                best_manager, best_amt = resolve_competition(manager_offers)

        # Assign if feasible
        if best_manager.can_buy(not_assigned[pid],
            best_amt):
                player_obj = not_assigned[pid]
                player_obj.assigned_to = best_manager.
                    name
                player_obj.final_price = best_amt
                best_manager.update_roster(player_obj,
                     best_amt)
                del not_assigned[pid]

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

Listing 4: 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:

```
def forced_assignments(managers, leftover_players):
        for mgr in managers:
                for role, (min_r, max_r) in mgr.
                    role_constraints.items():
                        count_current = sum(p.role ==
                             role for p in mgr.team)
                while count_current < min_r and
                    leftover_players:
                # pick any leftover of that role for 1
                     credit
                candidate = None
                for lp in leftover_players:
                        if lp.role == role:
                                candidate = lp
                                break
                if not candidate:
                        break  # no more players of
                             that role
                if mgr.can_buy(candidate, 1):
                        candidate.assigned_to = mgr.
                             name
                        candidate.final_price = 1
                        mgr.update_roster(candidate,
                             1)
                        leftover_players.remove(
                             candidate)
                        count_current += 1
```

Listing 5: Simple forced assignment to fill leftover roles

4 Example: Initial Inputs and Scoring Calculation

In this section, we present an overview of the initial data and results used in our optimization process. We include several graphical and tabular analyses to illustrate the diversity of input data and the effectiveness of our 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 our 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 is essential because it confirms that our experiment has a balanced input—each strategy is well-represented. A balanced manager distribution promotes diverse exploration in the search space, which is crucial for the robustness of the algorithm.

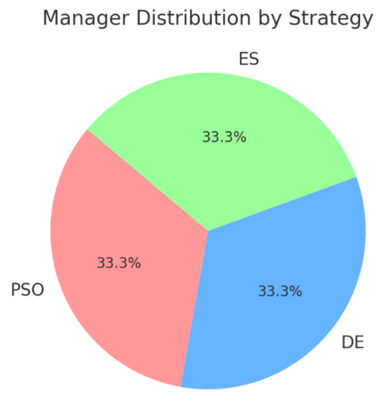

Figure 1: Distribution of managers by strategy.

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 our algorithm is demonstrated by its ability to select the optimal balance between expensive top performers and numerous affordable players.

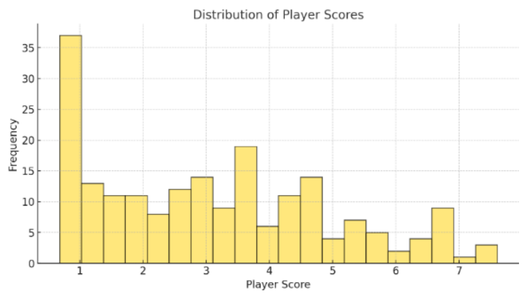

Figure 2: Distribution of player scores based on historical performance.

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 resources. Effective budget usage is critical in ensuring that no valuable credits remain unspent, thus maximizing the potential overall team score.

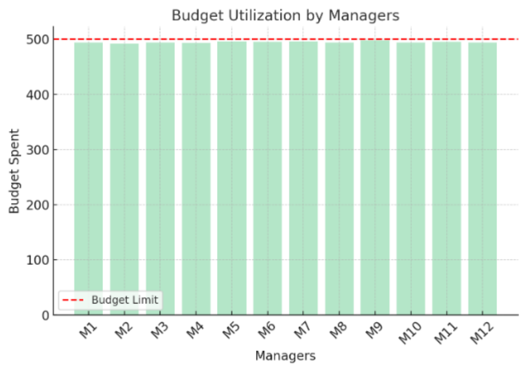

Figure 3: Budget Usage by Managers.

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.

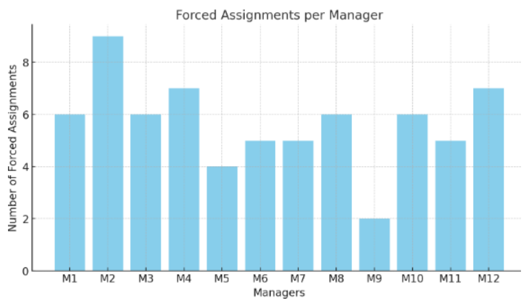

Figure 4: Number of Forced Assignments per Manager.

4.1.5 Average Team Score by Strategy. Finally, Figure 5 shows the average team score achieved by managers using each strategy. The graph demonstrates that while DE typically achieves a higher total score, PSO and ES also perform competitively. The relatively small differences among strategies suggest that the algorithm is robust, as it can achieve good performance regardless of the chosen method.

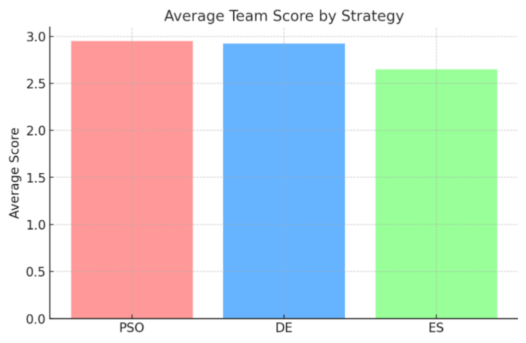

Figure 5: Average Team Score by Strategy (PSO, DE, ES).

4.2 Tabular Analyses

4.2.1 Manager Recap Table. Table 1 provides a detailed recap of each manager's performance. It includes the chosen strategy, the number of forced assignments, the total budget spent, leftover credits, and the final objective score (team score). A low number of forced assignments along with near-zero leftover budget demonstrates that the algorithm effectively satisfies all constraints.

Table 1: Recap Table: Manager Stats

Manager	Strat	Forced	Spent	Leftover	Objective
Manager_1	PSO	6	494.0	0.0	52.91
Manager_2	DE	9	492.0	0.0	51.48
Manager_3	PSO	6	494.0	0.0	44.46
Manager_4	DE	7	493.0	0.0	53.64
Manager_5	ES	4	496.0	0.0	49.18
Manager_6	ES	5	495.0	0.0	46.02
Manager_7	PSO	5	496.0	0.0	47.00
Manager_8	DE	6	494.0	0.0	55.32
Manager_9	ES	2	498.0	0.0	44.23
Manager_10	DE	6	494.0	0.0	46.59
Manager_11	ES	5	495.0	0.0	39.99
Manager_12	PSO	7	494.0	0.0	53.23

4.2.2 Performance by Strategy Table. Table 2 aggregates the performance of each strategy, reporting the number of managers using each method along with the average total score and average team 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 2: Performance by Strategy

Strategy	Managers	Avg Total Score	Avg Team Score
PSO	4	49.40	2.95
DE	4	51.76	2.92
ES	4	44.85	2.65

4.2.3 Player Score Summary Table. Table 3 summarizes the overall range of player scores calculated by our 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 our multi-objective optimization.

Table 3: Player Score Summary

	Best	**Worst**	**Average**
Player Scores	12.90	0.68	2.80

4.3 Overall Discussion

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

- The balanced manager distribution (Figure 1) ensures diverse search behavior.
- 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 (Figure 5 and Table 2) confirm that our optimization process consistently produces competitive teams.
- The player score summary (Table 3) emphasizes that only a few players reach elite performance levels, reinforcing the need for strategic selection.

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

5 Conclusion and Future Directions

This student 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** discovered decent lineups very fast but sometimes left leftover budget.
- **DE** refined solutions precisely, often maximizing the entire budget.
- **ES** preserved diversity, occasionally stumbling onto interesting rosters that others ignored.

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 Lineup Suggestions**: Extend the system to not only propose a full roster but also suggest an optimal starting lineup based on current form and projected performance, thus assisting managers in making final tactical decisions.

Even as a student implementation, the results show that *metaheuristics + 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.