

**Master Programme**

**Heuristic Optimization Methods**

REPORT - Lab1  **Fantasy football draft problem**

**Adrian Sušec**

**Ac. year 2023/2024**

**Contents**

1. Summary of best-found results 3
2. Greedy algorithm 4
   1. Pseudocode 4
   2. Description 4
   3. Analysis 4
3. GRASP 5
   1. Pseudocode 5
   2. Description 5
   3. Analysis 6

# Summary of best-found results

**Instance 1**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Score | First team lineup | Substitutions |
| Greedy algorithm | 1550 | 87,112,10,430,363,636,364,409,156,138,681 | 77,332,759,758 |
| GRASP | 1829 | 360,363,364,668,342,87,669,430,10,112,102 | 77,332,333,759 |

**Instance 2**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Score | First team lineup | Substitutions |
| Greedy algorithm | 1325 | 16,139,119,72,445,74,118,598,559,554,344 | 70,521,522,523 |
| GRASP | 1550 | 16,139,118,72,119,554,285,282,545,551,73 | 70,521,522,523 |

Programming language: Kotlin

# Greedy algorithm

## Pseudocode

firstTeam = []

substitutions = []

sortedPlayers = players.sortByPointsPerPriceDescending()

**while** size(firstTeam) < 11:

player <— selectFirst(sortedPlayers)

**if** (player not breaking constraints):

firstTeam <— firstTeam ∪ player

sortedPlayers = players.byPriceAscending()

**while** size(substitutions) < 4:

player <— selectFirst(sortedPlayers)

**if** (substitutionPlayer not breaking constraints):

substitutions <— substitutions ∪ substitutionPlayer

**return** firstTeam, substitutions

## Description

* *Clarify the greedy selection heuristic. How are elements selected when adding them to the solution?*

The greedy selection heuristic calculates the points per price for a certain player. The point of it is to try to select a good player that would allow us to spend the least possible amount of budget while still fetching a solid amount of points.

The elements(players) are selected by max points per price, meaning that the player with the best points per price available is selected (if it doesn’t break the constraints).

## Analysis

* *If you tried different greedy criteria, comment and compare results.*
* *How could the greedy algorithm potentially be improved?*

Players with best points per price are usually defenders and midfielders, while the attackers are more expensive meaning the algorithm will take the midfielders and defenders first, and only select the attacker when it needs to satisfy the constraints.

Potential improvements could be made by selecting a different criteria (e.g. multiplying the price by some factor alpha and points by factor beta, which would give us the ability to place more importance on a certain factor. Also, enforcing certain formations to fulfil might should give a better team considering attackers carry more points, which would also lead to better budget exploitation.

# GRASP

## Pseudocode

**GRASP Pseudocode**

**for** i in **range**(0, 10):

seededTeam <— greedyRclAlgorithm(alpha)

bestNeighbour <— localSearch(seededTeam)

update(currentBest, bestNeighbour)

**return** currentBest

**GRASP Construction Phase**

**while** first team size < 11:

RCL = generateRCL(alpha, players)

player <— selectRandom(RCL)

**if** (player not breaking constraints):

firstTeam <— firstTeam ∪ player

subs <— generateSubsWithCheapestPlayers(players)

**return** firstTeam, subs

**GRASP Local Search Phase**

**do**

neighbours <— generateNeighbours(currentFirstTeam)

*evaluate neighbours*

**for** neighbour in neighbours:

**if** fitness(neighbour) > fitness(current):

update(neighbour, current)

didUpdate <— true

**while** didUpdate is true

## Description

* *Explain both the constructive and local search phases.*

The constructive phase works similar to the greedy algorithm, with the “only” difference being the RCL. The local search phase takes the previous best solution and generates a neighbourhood for it. Then it searches through the neighbourhood and selects the best possible solution and replaces the previous best in case it is better.

* *How is the RCL obtained?*

Instead of taking the first best candidate, the algorithm takes a random candidate from the RCL, which is generated according to the formula available in the slides. The size of the RCL is controlled by the hyper-parameter alpha, which is controlled by the user (between 0 and 1).

* *During local search, how are neighboring solutions generated?*

Neighbouring solutions are generated by iterating over the list of players in the team, and trying to replace each players with 10 best available players for that position, if it adheres to the constraints. It works as a single swap (each neighbour has only one player different to the original team that the neighbourhood was generated around).

* *Can you comment on neighborhood size? What is the complexity of your algorithm?*

Neighbourhood size is at most 110 (replacing each player with at most 10 other players).

Construction phase -> O(N) where N is the size of the team

Local search phase -> O(N) where N is the size of the team - neighbourhood generation part

O(N) where N is the size of the team - searching the neighbourhood

-> no of neighbours <= 10x team size

-> still linear with respect to team size

## Analysis

* *Was there any benefit from using an RCL containing multiple solution elements, as opposed to simply using your original greedy algorithm followed by local search?*

Most definitely, the solution generated for instance 1 was around 250 points better and it also used up the available budget to the max. Similar impact was seen on instance 2.

* *Was there any impact of RCL size on solution quality? If yes, quantify or plot these values.*

The biggest difference that was noticed between the alpha being >= 0.3 and alpha being < 0.3. Solutions with alpha 0.3 for RCL were better than the solution generated when alpha < 0.3. The most common no of points with alpha < 0.3 was 1761, while the randomness started becoming more obvious with alpha >= 0.3. Solutions with alpha around 0.5 were unstable, meaning they could give either really bad or really good solutions. Alpha 0.7 proved to be quite good, with solutions usually being over 1800 points.

* *How many iterations of the local search algorithm were needed to reach a local (or potentially global) optimum?*

Around 10 iterations (+- 2) in most cases.

* *Do you have any ideas for different neighborhoods that you could use?*

There is a possibility of generating all possible teams which would be a combinatorial nightmare. Instead of a single swap, we could generate teams with a double, triple or a bigger number of swaps. We could also change with the players from the same team.

* *How do you expect your algorithm would perform for much larger instance sizes?*

Time complexity of the algorithm should not be a problem since the neighbourhood size is growing linearly with the amount of players added. The neighbourhood is generated by taking player which should be better, which allows us to use up more of the budget. Knowing that the algorithm in the construction phase is greedy, meaning the initial solution should be generated quite quickly, the complexity is O(N) where N is the size of the team that needs to be filled up.