

**Master Programme**

**Heuristic Optimization Methods**

REPORT - Lab1  **Fantasy football draft problem**

**Antoine Vieville**

**Ac. year 2023/2024**

**Contents**

1 Summary of best-found results 3

2 Greedy algorithm 4

2.1 Pseudocode 4

2.2 Description 4

2.3 Analysis 4

3 GRASP 4

3.1 Pseudocode 4

3.2 Description 4

3.3 Analysis 4

# Summary of best-found results

(I have included results obtained by a more general greedy algorithm, I explain my thinking process in 2.2)

**Instance 1**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Score | First team lineup | Substitutions |
| Greedy algorithm | 1787 | 668,669,363,342,360,347,87,10,430,156,157 | 76,332,333,759 |
| (General greedy algo) | (1602) | (87,112,10,430,363,636,409,360,156,681,680) | (76,332,333,759) |
| GRASP | 1639 | 138,343,102,101,681,347,668,409,360,680,10 | 289,226,498,24 |

**Instance 2**

|  |  |  |  |
| --- | --- | --- | --- |
|  | Score | First team lineup | Substitutions |
| Greedy algorithm | 1492 | 285,282,72,545,543,16,139,119,445,344,326 | 70,271,272,626 |
| (General greedy algo) | (1273) | (16,139,119,72,445,598,559,344,370,326,313) | (70,271,272,626) |
| GRASP | 1486 | 139,551,8,285,545,73,72,344,326,554,290 | 541,237,271,67 |

Programming language: Python

# Greedy algorithm

## Pseudocode

**Begin** greedy algorithm (*hybrid version*)

**Input** : instance, n top players of the instance

Initialize an empty bench

Initialize an empty team (on the field)

**Repeat** :

Draft cheapest player while keeping feasibility (meeting the club/position/budget constraints)

**While** bench is not full

**Repeat** :

Draft top player of the instance while keeping feasibility

**Until** we have drafted n top players

**Repeat** :

Draft player based on its ratio points/price while keeping feasibility

**While** team on the field is not full

**End**

**Return** : team on the field, bench, budget spent on the draft

**Begin** greedy algorithm (*general version*)

**Input** : instance

Initialize an empty bench

Initialize an empty team (on the field)

**Repeat** :

Draft cheapest player while keeping feasibility (meeting the club/position/budget constraints)

**While** bench is not full

**Repeat** :

Draft player based on its ratio points/price while keeping feasibility

**While** team on the field is not full

**End**

**Return** : team on the field, bench, budget spent on the draft

## Description

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

My first approach was to :

* select 4 cheapest players of the instance as substitutes, among which 1 GK and 2 DEFs, which leaves more high scoring players like MIDs and FWs on the field
* select 11 best scoring players of the instance, filling the remaining positions.

However, with this approach, we run out of money, and we cannot build a team of 15 players. Indeed, there is no back tracking so you can’t remove a player that you already added to the draft. I had to think of a different approach.

My second approach (**general greedy algorithm**) was to :

* again select 4 cheapest players
* select 11 best players based on the ratio points/price.

With this approach, you can build a feasible and decent solution to the problem. However, for instance 1 for example, the budget spent was around 80, which means there were 20 left unused that could have been used to buy better players.

This made me go with a third a final approach : a hybrid of the first two approaches (**best scoring greedy algorithm**). I figured that with unspent money when taking the second approach, we could afford to buy a certain number of top players before resorting to the ratio criterion for selection.

* again select 4 cheapest players
* select n top players (n depends on the instance)
* select the rest of the players based on their ratio.

To find n in each instance, I passed this number as an argument and tried to increase until the algorithm wouldn’t run : this means with have reached the maximum of top players we can afford for this instance. I am aware that this approaches requires you to “know” the instance and to run the greedy algorithm multiple times on the instance. *I don’t know if it still qualifies as a greedy approach*, but the solutions built with it are feasible. If no information is known about the instance or it is too important to afford to try and find this number n, we can resort to the second approach (more general greedy heuristic). It will return a feasible but sub optimal solution.

## Analysis

* *If you tried different greedy criteria, comment and compare results.*

Most points criteria : when using this criteria, even when assign 4 cheapest players as substitutes, the budget constraint can never be met.

Best ratio points/price criteria : gives you a pretty good solution but suboptimal. You can see changes that could be made by hand with the remaining budget. When trying this, the best team score I got for instance 1 was 1602 for example.

The hybrid criteria I used gave me the better score with the best use of the budget, but it requires you to know the number of top players you can afford in each instance.

instance 1: score = 1787, budget = 100 ;

instance 1: score = 1492, budget = 99.6 ;

* *How could the greedy algorithm potentially be improved?*

Maybe there is a way of achieving the hybrid approach I took without prior knowledge of the instance (n top players), while still respecting the no backtracking rule. I haven’t found a way to do this.

When I pick substitutes, I assume that there should be 2 DEFs on the bench because most of the top players are MIDs and FWs. Maybe picking one of each positions can give better scores, considering some top DEFs score a lot of points too.

# GRASP

## Pseudocode

Construction phase :

**Begin** construction phase

**Input** : instance, seed

Initialize empty bench

Initialize empty team (on the field)

**Repeat** :

Draft cheapest player while keeping feasibility (meeting the club/position/budget constraints)

**While** bench is not full

**Repeat** :

Build RCL : all players whose points are superior or equal to the threshold

Draft a player from the RCL at random while keeping feasibility and remove from instance

Reevaluate players of the instance based on points

**While** team on the field is not full

**End**

**Return** team on the field, bench, budget spent on the draft

Local search :

**Begin** local search

**Input** : instance, draft, (neighborhood size, default =10)

**While** a better team can be found (solution is not locally optimal)

**For** each player in the current team

Generate the player neighborhood : ten random players of the same position

Replace player by the best improving neighbor (better scoring player, if any are found) while keeping feasibility

**End**

**Return** better team (optimal solution), bench, budget spent on the draft

## Description

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

In the constructive phase, we randomized the selection of players while keeping a greedy approach. It is safe to assume that the substitutes should still be the cheapest players, since they will allow us to save to money to spend on better players. Thus I haven’t randomized this part. Then, the drafting of the 11 players is randomized by the RCL but still greedy in the sense that players are evaluated based on their ratio (we take “one of the best” at each iteration). This phase gives us a randomized feasible but sub optimal solution.

In the local search, I didn’t try to improve the substitutes, considering the search will be run on a draft generated by the construction phase, and this phase already drafts cheapest substitutes. They shouldn’t need to be changed. Then, for each player in the current playing team, I define a neighborhood of the current solution : one player switch from the current team, with the same position. I explore this neighborhood searching for better solution (reduced random neighborhood and picking best improving neighbor ie switching for a better player). We continue until we cant improve our solution anymore.

* *How is the RCL obtained?*

In theory, the RCL is only composed of elements that can keep feasibility. However in this problem, checking all feasible players for on given draft (in construction) would mean to go over every players of the instance and check if we can add them, at each run until the draft is complete. I thought that was not optimal (especially if the instance is important) and that a RCL composed not only of draftable players would do, with the condition that we keep the RCL size decent (not too small or we might be in the case where none of the RCL players are draftable, ie alpha not to close to 1). Knowing this, here is how I built my RCL : I set the threshold with the formula c\_min + alpha\*(c\_max-c\_min), take all players who have more points than this threshold as RCL. Then I shuffle this RCL and I take the first player I see that meets the constraints. Because of the threshold, the RCL contains “the best elements”, and the shuffle assures that the player drafted is chosen at random.

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

The neighborhood explored is 10 random players (set neighborhood size, is a default argument of the algorithm but can be changed) playing the same position as the player of the iteration. If among these 10 random players, we find any feasible and improving replacements/switches, we store them. We search the most improving neighbor among these stored and replace him with the current player. I took this randomized neighborhood approach so that we don’t get stuck in a locally optimal solution but explore all possible switches.

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

A neighbor solution is generated by switching one player from the current team. Therefore, the neighborhood size is less or equal to 10. A larger neighborhood would maximize chances to find better players, but requires more exploration. A smaller neighborhood reduces the chances of finding better players but requires less exploration. We run this phase until no better solution is found, so the complexity of the algorithm would be O(neighborhood\_size \* nb of iterations needed).

## 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?*

When comparing to the general greedy algorithm (fully based on ratio points/price), having multiple players to choose from and picking one randomly leaves you with more freedom in the team composition (we don’t always pick the same players in the same order and then try to improve them) and can lead to better results when you apply local search on it.

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

With my implementation, since there is no backtracking, if we set to RCL size too low, we might end up in a situation where we cannot find any players meeting the constraints, and therefore the algorithm will never converge (because no back tracking is possible, we get stuck). Unfortunately, I couldn’t find an other approach to solve this problem.

However, if we remain in acceptable range of alphas (say alpha <=0.8), the RCL becomes more “selective” of the very best players and thus the team drafter will be of better quality (higher fitness). I plotted the fitnesses of solutions for different values of alphas for instance 1 :

Une image contenant capture d’écran, affichage, texte, logiciel

Description générée automatiquement

*Une image contenant texte, capture d’écran, affichage, logiciel

Description générée automatiquement*

Because there is randomness evolved in the choice of “one of the best” elements, the increase in the fitness solution is not necessarily consistent but we can see that there is a trend.

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

For the ouput of GRASP, local search for the best solution required :

13 iterations for instance 1, 26 iterations for instance 2

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

Instead of switching one player with another player from the same position, we could have kept this idea of switching but with any better player. This way, we could witness different team composition. This is a bit more complicated to implement because we have to check the position constraint for each improving neighbor, every time a player is switched (position constraint applied to the new team).

Another idea would be to randomly switch 3 players from the bench, while keeping 1 GK. This way, there might be 2 DEFS 1 MID in one case, 2 MID 1 FW, 2 FW 1 DEF… One of these combinations might maximize the points scored by players on the field. I tried implementing this with randomly switching with cheap players but somehow the position constraint ended up being not met in the final draft. I couldn’t fix it.

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

Since my RCL is not composed of only viable players to draft, it doesn’t require to check the viability of all players of the instance at each iteration of the drafting. That saves some computational time on larger instances. For the local search, we could also afford to increase the neighborhood size in order to not run the algorithm on many iterations and have more chances to draft the best players early. Otherwise, if we keep the default value of 10 neighbors, the algorithm may take long to converge on larger instances.