UNIVERSITY OF ZAGREB
Faculty of Electrical Engineering and Computing

Master Programme

Heuristic Optimization Methods

REPORT - Lab1

Fantasy football draft problem

Bernard Crnković

Ac. year 2022/2023

Contents

1 Summary of best-found results

Instance 1

	Score	First team lineup	Substitutions
Greedy algorithm			
GRASP	1963	4,48,72,71,412,218,421,306,424,263,246	21,77,81,276

Instance 2

	Score	First team lineup	Substitutions
Greedy algorithm			
GRASP	1998	10,74,75,77,578,301,308,584,315,117,87	9,489,539,629

Instance 3

	Score	First team lineup	Substitutions
Greedy algorithm			
GRASP	1511	16,72,73,74,551,285,554,291,559,139,313	8,119,445,326

Note: I didn’t notice that I had to write down solutions from greedy approach. Since GRASP immediately followed first part of algorithm and could only improve solution further, I didn’t write down greedy scores, however, looking at the verbose output in console, one can check that in fact, solutions were worse using pure greedy approach.

Programming language: ___Python 3.10.8___

2 Greedy algorithm

2.1 Pseudocode

```
def generate_initial_solution(dataset: Dataset, max_iter=100):
    team = []
    current_iter = 0
    for position, n in [('GK', 2), ('DEF', 5), ('MID', 5), ('FW', 3)]:
        len_before = len(team)
        while n > len(team) - len_before:
            position_players = dataset.positions[position]
            weights = exp_weights(len(position_players))
            selection = random.choices(position_players, weights=weights, k=1)
            if valid_solution(team + selection):
                team += selection
            current_iter += 1
            if current_iter >= max_iter:
                return None
    return team
```

2.2 Description

- *When creating initial solution, players are sampled one by one from pool of all players sorted by their perceived quality (quality being expressed as points / price). When sampling, weights are assigned according to exponential function lambda i: 2**-i for i-th index in list of players. Solution is validated to check if solution has a potential to be valid. If not, process is restarted.*

2.3 Analysis

- *I tried changing certain parameters: weight function for sampling distribution (using linear function, or different base for lambda i: b ** -i), changing number of random restarts and choosing best built solution from multiple greedy runs. Also, changing max unsuccessful iterations when sampling players for a single greedy solution.*
- *I could probably sample more efficiently by sampling more players at once which avoids cases of trying invalid solutions where e.g. more than 2 ‘GK’ positions are chosen. In general, avoiding building impossible solutions would shorten the runtime of algorithm. However, after tweaking parameters for my computer, current runtime, which is around a minute was giving acceptable results.*

3 GRASP

3.1 Pseudocode

```
def grasp(dataset, original_solution, n_neighbourhood=1):
    original_score = score(original_solution)
    current_best = original_solution[:]
    current_best_score = original_score
    for solution_indices in combinations(range(len(original_solution)), n_neighbourhood):
        positions = tuple(original_solution[i].position for i in solution_indices)
        len_ranges = tuple(range(len(dataset.positions[p])) for p in positions)
        for sample_indices in product(*len_ranges):
            tmp_solution = original_solution[:]
            for sol_index, position, sample_index in zip(solution_indices, positions, sample_indices):
                tmp_solution[sol_index] = dataset.positions[position][sample_index]
            if not valid_final_solution(tmp_solution):
                continue
            tmp_score = score(tmp_solution)
            if tmp_score > current_best_score:
                current_best = tmp_solution
                current_best_score = tmp_score
    return current_best
```

3.2 Description

- *Explain both the constructive and local search phases.*
- *In construction phase, I generate any feasible solution without any lookahead, just currently highest ranked player by custom criteria. In search phase, I take generated (valid) solution and try to do substitutions in n-neighboorhood (tweakable) around my solution.*
- *How is the RCL obtained?*
- *RCL is obtained by grouping dataset according to different criteria (pre-computed and reused, e.g. player position bins, club bins), which significantly reduces search space. I tried running without criteria in brute force approach, but combinatorial explosion quickly became a problem. Later, semi-brute-force approach worked fine for 2-neighboorhood.*
- *During local search, how are neighboring solutions generated?*
- *semi-brute-force, meaning, constrained search space is brute forced.*
- *Can you comment on neighborhood size? What is the complexity of your algorithm?*

- *I checked only 2-neighboorhood because I don't have strong CPU, and didn't manage to extract better performance from Python. It is quite difficult to tell what complexity would be, probably in O(n^4) range with very small constant in front, because search space doesn't have 'simple shape'.*

3.3 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?*
- *Yes, solution space probably has many 'peaks and valleys' but I can't prove it. Empirically though, It is quite noticeable that different starting seed yields quite different results. Sampling more in the beginning improves chances of finding better local optimum with GRASP later.*
- *Was there any impact of RCL size on solution quality? If yes, quantify or plot these values.*

- *I didn’t notice significant effect on solution quality. Solutions with larger RCL were slightly better but not enough reruns for a plot.*
- *How many iterations of the local search algorithm were needed to reach a local (or potentially global) optimum?*
- *Since my implementation uses pre-determined number of iterations for greedy step, and then brute-force approach on neighbourhood exploration (checking everything in range), number of iterations is tweakable.*
- *Do you have any ideas for different neighborhoods that you could use?*
- *We could define for example price-neighbourhood, where we would sample substitutes for player in some position from certain price/point range but from different teams. The benefit that this may give us is that we would ‘jump’ to different location in search space without significantly affecting quality of our current solution.*
- *How do you expect your algorithm would perform for much larger instance sizes?*
- *Without detailed profiling (using python’s cprofile) to see where the most runtime is spent, and deeper analysis, followed by refactoring, it is hard to say. Since there are heterogenous filtering criteria and constraints placed on a team, runtime may vary significantly depending on which data is in the input dataset.*