Algorithmic Methods for Mathematical Models
Course Project

Ignacio Encinas Rubio, Adrián Jiménez González

November 24, 2022

Contents

1 Problem statement

In this section we'll state the inputs, outputs, modelling elements and objective function.
The problem can be summarized into the following requirements:

1. Each contestant will play exactly once against each of the other contestants.
2. Each round will consist of $\frac{n-1}{2}$ matches.
3. Players will play 50% of their games as white, 50% will be played as black.

1.1 Input

- n is the number of players that will participate in the tournament.
- $p_{n \times n}$ is the matrix of points. p_{ij} is the number of points assigned by player i to round j.

1.2 Definitions

We've defined the following sets in order to model the problem:

- P is the set of players participating in the tournament. $|P| = n$
- Rounds is the number of rounds. In this case it is equal to $|P|$.
- $M(x, y)$ is the set of matches played among x and y.
- $F(r)$ is the set of free players at round r.
- $W(p)$ is the set of matches played by player p as white.
- $B(p)$ is the set of matches played by player p as black.
- $R(r)$ is the set of matches played at round r.
- $G(p, r)$ is the set of games played by player p at round r.

Thus, the score of a schedule can be computed as followed:

$$\text{Score} = \sum_{j=1}^{\text{Rounds}} \sum_{i \in F(j)} p_{ij} \tag{1}$$

Our goal is to maximize this score.

1.3 Output

The output will be an optimal schedule s that maximizes the score. It will contain the matches to be carried out in every round of the tournament.

2 Integer Linear Programming Model

Every set will be built from a boolean multidimensional array. $matches[w][b][r]$ will be 1 whenever player w[1] plays player b[2] in round r, and 0 otherwise.

For clarity's sake we'll explicitly state these constructions:

$$
\begin{aligned}
M(x,y) &= \{\{x,y,r\} && r \in [1, Rounds] \mid \text{matches}[x][y][r] = 1 \vee \text{matches}[y][x][r] = 1 \} \\
F(r) &= \{p && o \in [1,n] \mid \text{matches}[p][o][r] = 0 \wedge \text{matches}[o][p][r] = 0 \} \\
W(p) &= \{\{p,b,r\} && r \in [1, Rounds], b \in [1,n] \mid \text{matches}[p][b][r] = 1 \} \\
B(p) &= \{\{w,p,r\} && r \in [1, Rounds], w \in [1,n] \mid \text{matches}[w][p][r] = 1 \} \\
R(r) &= \{\{w,b,r\} && w,b \in [1,n] \mid \text{matches}[w][b][r] = 1 \} \\
G(p,r) &= \{\{o,p,r\} && o \in [1,n] \mid \text{matches}[p][o][r] = 1 \vee \text{matches}[o][p][r] = 1 \}
\end{aligned}
$$

The objective function was already described by Eq. 1

2.1 Constraints

$$|M(x,y)| = 1 \quad \forall x,y \in P \mid x \neq y \tag{2}$$

Eq. 2 ensures "each contestant will play exactly once against each other of the contestants".

$$|G(p,r)| \leq 1 \quad \forall p \in P, r \in [1,\ \text{Rounds}] \tag{3}$$

Eq. 3 ensures that a player plays up to 1 game per round.

$$|R(r)| = \frac{n-1}{2} \quad \forall r \in [1,\ \text{Rounds}] \tag{4}$$

Eq. 4 ensures that "the number of games that are played simultaneously at each slot is always $\frac{n-1}{2}$"

$$|W(p)| = \frac{n-1}{2} \quad \forall p \in P \tag{5}$$

Eq. 5 ensures that "a contestant should play black as many whites as white", given that Eqs. 2, 3 and 4 ensure that $|\text{Games}(p)| = n-1$ and $|\text{Games}(p)| = |W(p)| + |B(p)|$

2.2 Redundant constraints

$$|M(x,x)| = 0 \quad \forall x \in P \tag{6}$$

Eq. 6 ensures that a player can't play with himself.

$$|B(p)| = \frac{n-1}{2} \quad \forall p \in P \tag{7}$$

While developing the model, we removed some redundant constraints such as the one described by Eq. 6. Initial testing suggested that the redundant ILP model could be faster in some cases, but further testing indicated otherwise. Results shown in Figure 1

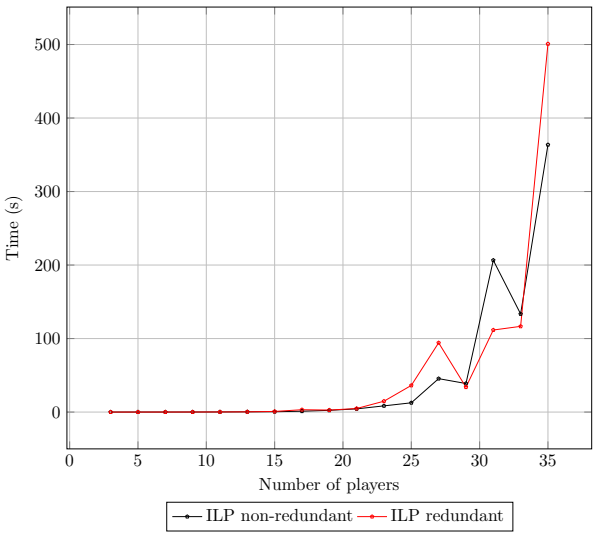

Figure 1: ILP runtime w.r.t the number of players

[1] Player with the white pieces
[2] Player with the black pieces

3 Meta-heuristics

As a solution must exist no matter which player rests in a given day, we will focus on assigning a rest day for every player p. This information is enough for calculating our objective function, performing local search and GRASP. After all this, we will determine the pairings for the whole tournament.

3.1 Constructive

The greedy cost function is the points that a player p assigns to resting on day d. As we want to maximize this, we will sort the available players in descending order and pick the one with highest value.

$$q(c, day) = c.points\_per\_day[day] \tag{8}$$

Eq. 8 is used as greedy cost function to take who rests.

Algorithm 1 Greedy algorithm

```
1: Players ← Set of Players
2: rests ← {}
3: for day in 0..days do
4:     playersToRest ← filter Players(p) | p.hasNotRested
5:     sortedPlayers ← sort playersToRest(p) by q(p,day) (DESC)
6:     select p ∈ sortedPlayers.first()
7:     rests[day] ← p
8: end for
```

3.2 Local search

In the local search phase we evaluate if swapping the players that rest in two given days increases our objective function[3].

Algorithm 2 Local Search

```
1: for i in 0..days do
2:     best_swap_points ← 0
3:     best_swap ← i
4:     for j in 0..days do
5:         change = EvaluateRestSwap(i,j)
6:         if change > best_swap_points then
7:             best_swap_points ← change
8:             best_swap ← j
9:         end if
10:    end for
11:    rests[i] ↔ rests[best_swap]
12: end for
```

We can repeat this procedure iteratively until the local search provides no further improvement just by adding an outer loop.

[3]For simplicity, we initialize the best_swap so that the local search doesn't change the solution if we found no improvement

3.3 GRASP

In the GRASP we randomize the selection of the player in descending order by points, according to a RCL calculated by α as shown in Algorithm 3.

Algorithm 3 constructRCL(day)

1: $q_{max} \leftarrow$ sortedPlayers.first().points[d]
2: $q_{min} \leftarrow$ sortedPlayers.last().points[d]
3: $RCL_{max} \leftarrow \{p \in sortedPlayers \mid p.points[d] >= q_{max} - \alpha \cdot (q_{max} - q_{min})\}$

Algorithm 4 GRASP

1: rests $\leftarrow$ {}
2: **for** day in 0..days **do**
3: playersToRest $\leftarrow$ filter Players(p) | p.hasNotRested
4: sortedPlayers $\leftarrow$ sort playersToRest(p) by q(p,day) (DESC)
5: RCL $\leftarrow$ constructRCL(day)
6: select p $\in$ RCL randomly
7: rests[day] $\leftarrow$ p
8: **end for**

Our stopping criteria combines the number of iterations since the last incumbent update and a fixed maximum number of iterations.

3.3.1 Parameter tuning

For tuning α, we have compare GRASP's objective function with the optimal solution for each of the instances. Then, for each α we calculate the arithmetic mean error.

Figure 2 shows the mean error of every α we have tested against instances $\{0 \dots 31\}$. Two separate lines are plotted for $\alpha_i = 0 + i \cdot 0.1$ and $\alpha'_i = 0 + i \cdot 0.01$. Results show that many values for alpha give good results. We decided to keep the smallest alpha that gives the minimum mean error, so $\alpha = 0.19$ is the chosen value after tuning.

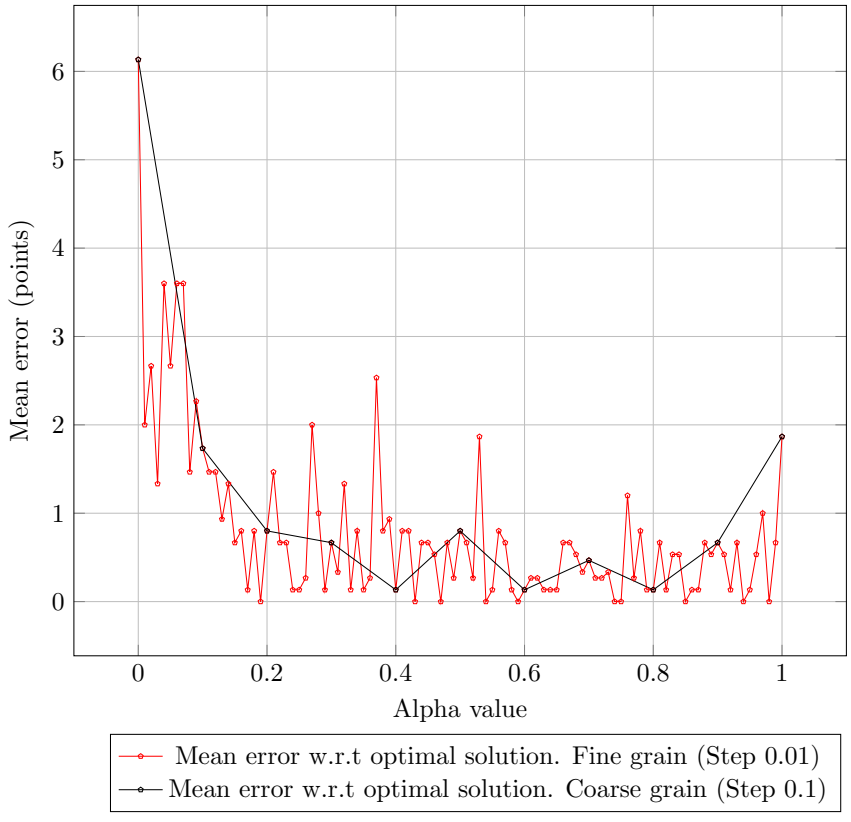

Figure 2: Mean error w.r.t alpha

4 Results

4.1 Time

Figure 3 shows the runtime needed for the different models and solvers. Section 2.2 introduced the redundant and non-redundant ILP models. ILP rest corresponds to the ILP model that just computes the optimal rest day for each player, without obtaining a valid tournament schedule.

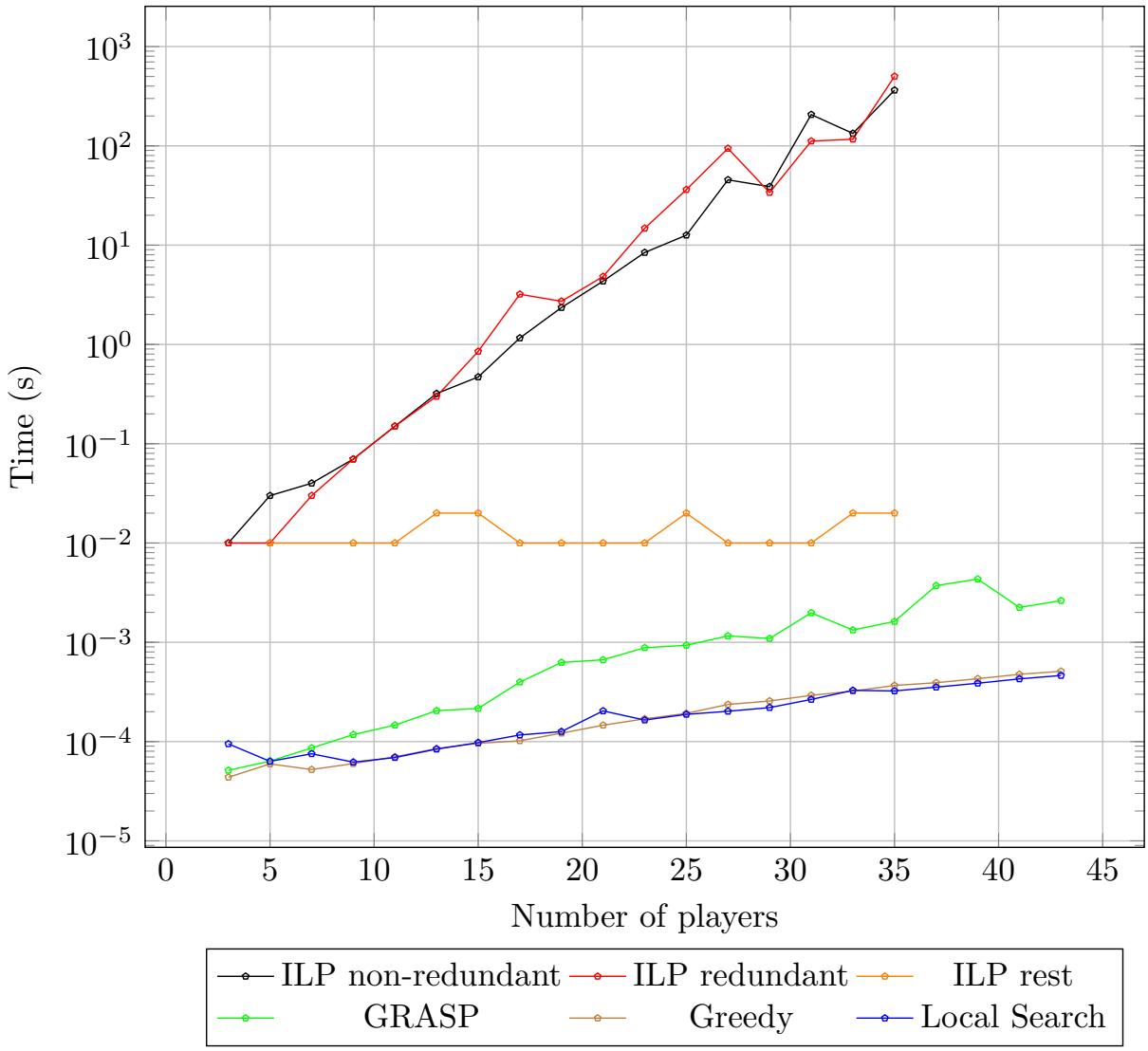

Figure 3: Runtime for the different ILP models and (meta)heuristic based solvers

Greedy and Local Search need approximately the same time to reach the solution. They are several orders of magnitude faster than GRASP or the ILP models, but the quality of the solution is too low.

GRASP sacrifices some of the speed in order to improve the quality of the solution, that is generally acceptable. It is also true that the runtime is highly dependant on the stopping criteria and could be tuned for specific requirements if needed.

Our motivation for testing this simpler ILP model is that while solving the assignment we realized that the hardest part of it was actually getting a valid tournament schedule, not optimizing the objective function. Then, we started wondering whether the CPLEX time cost was spent optimizing the objective function or not.

Results indicate that our minimal CPLEX model is very fast, and we could combine its output with a tournament schedule maker (see Section 5) to obtain the optimal solution several orders of magnitude faster than the complete CPLEX model while maintaining optimality, in constrast to heuristic-based solvers. For large instances the simple ILP model is even faster than our GRASP implementation[4].

[4]Of course if the stop criteria plays a great role in the GRASP runtime, and it could be greatly reduced if we're willing to tolerate worse solutions

4.2 Objective function

Figure 4 shows the value of the objective function obtained for each of the implementations: ILP, Greedy, Local Search and GRASP.

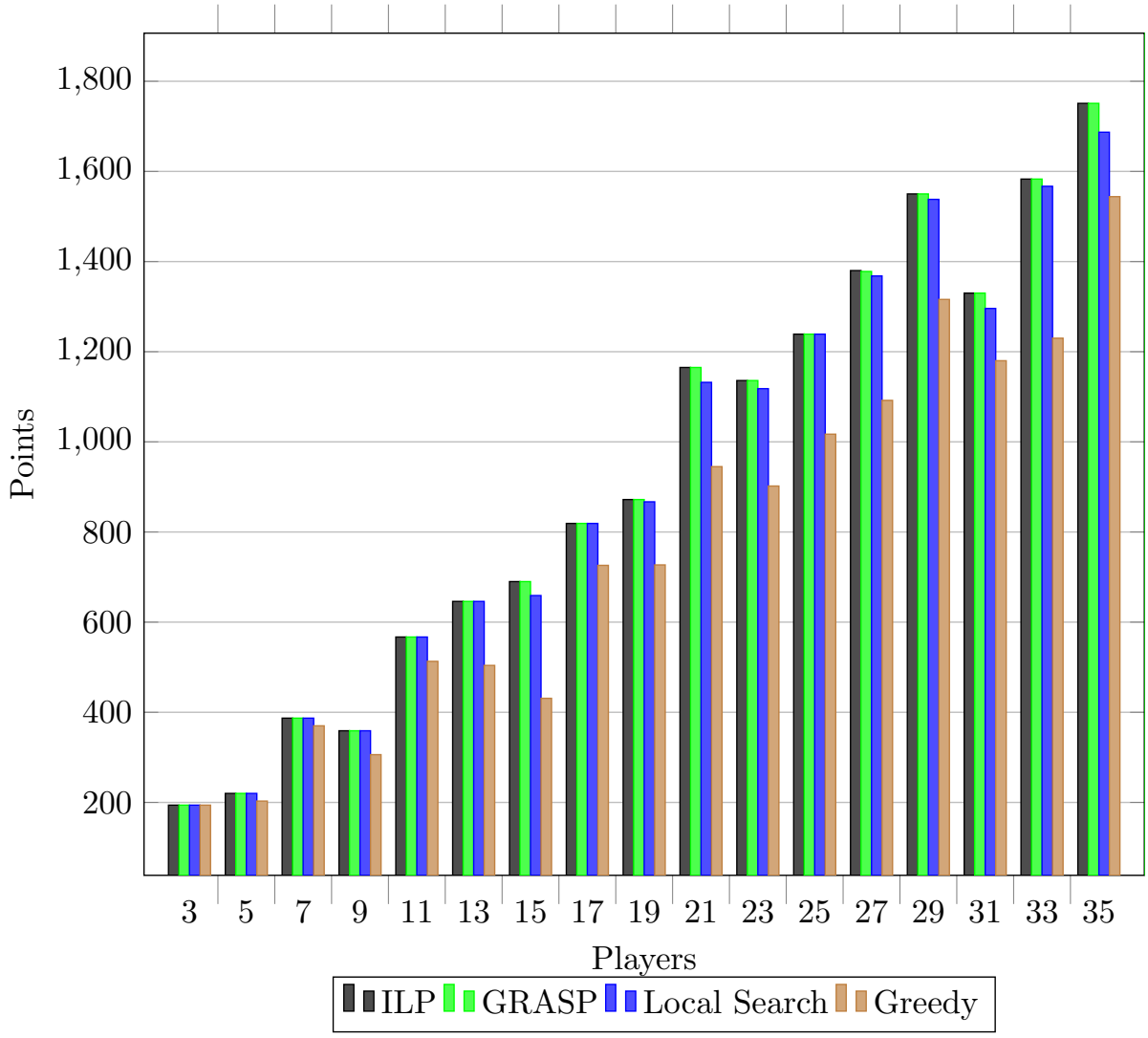

Figure 4: Objective function value per instance and solver

Naturally, greedy offers the worst solution for every instance, while taking practically the same time as Greedy + Local Search. Then, we can conclude that the basic Greedy method shouldn't be used in any case.

Local Search offers great improvement for every instance, but the quality of the solutions is noticeably lower than GRASP's.

GRASP commonly reachs the optimal solution. It obviously does not find it every time but if we're not too concerned with optimality is a good improvement over Local Search.

5 Schedule construction

When solving the assignment we assumed that the schedule construction would be trivial, and tried to construct it while optimizing the objective function. After many fruitless efforts, we gave up on the idea and decided to make our greedy solver just compute the rest day for each player. This is enough information to compute the objective function, as the actual pairings don't affect it.

We believe that the solution to this part of the assignment should have been provided to us, as it is much more difficult that the actual assignment and has nothing to do with the subject's contents. Even more so considering this was completely accidental.

Just for completeness we will briefly explain our solution and our thought process:

1. The problem with making the pairings are the rest days
2. If there were no rest days and we had an even number of players, obtaining a valid pairing is relatively simple

Then, our solution involves adding every day an extra invalid match for our players. This match can be a match against themselves or against an extra "fake" player. After doing this, we have what we wanted: a tournament schedule. The problem with this schedule is that it doesn't take into account our objective function and contains invalid matches, so for the moment it remains useless for us.

5.1 Solution

The trick is that we can use this template schedule to construct our desired schedule. The procedure is the following:

Our template schedule t will contain an invalid match each day. This invalid match is the one of the form p - p or p - f. Where p stands for player and f for "fake" player. From t we can extract what we call a *rest vector*[5] r, where player will rest a given day if he plays an invalid match. Keep in mind that our greedy algorithm provides us with our *desired rest vector* d.

Then, from t we can obtain a valid schedule that respects our desired rest vector d by removing invalid matches and remapping the players in the following way:

$$r[i] \leftarrow d[i] \quad \forall i \in [1, Rounds] \tag{9}$$

The explanation is a bit vague on purpouse for brevity's sake. The whole procedure can be better understood by reading the source code and in the toy example we'll describe next.

5.2 Toy example

Let's say we want to find a valid[6] schedule for a tournament of 3 players. Our template schedule looks like this[7]:

Day		
0	1	2
0-2	1-0	2-1
1-1	2-2	0-0

Table 1: Template schedule

The *rest vector* r would be:

$$r = \{1, 2, 0\}$$

While we want to be able to have an arbitrary *desired rest vector* d. For example:

$$d = \{2, 1, 0\}$$

Day		
0	1	2
0-1	2-0	1-2

Table 2: Valid schedule

Table 2 shows a correct schedule after remapping as indicated by Eq. 9:

$$1 \leftarrow 2,\ 2 \leftarrow 1,\ 0 \leftarrow 0$$

After remapping, our schedule will respect the *desired rest vector* d while also following the constraints.

[5] The rest vector just keeps track of which players rests which day

[6] The 50% white games and 50% black games can be easily added so we ignore the issue here, but it is obviously taken into account for the assignment

[7] The template schedule is arbitrary. See Wikipedia for more information