Algorithmic Methods for Mathematical Models
Course Project

Ignacio Encinas Rubio, Adrián Jimenez González

November 18, 2022

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:

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

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 plays player b in round r, and 0 otherwise.

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

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

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 $\frac{n-1}{2}$"

$$|W(p)| = \frac{n-1}{2} \quad \forall r \in [1,\ \text{Rounds}], \forall p \in P \tag{5}$$

Eq. 5 ensures that "a contestant should play black as many whites as white", given that Eq. 2 ensures 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.

$$|B(p)| = \frac{n-1}{2} \quad \forall r \in [1,\ \text{Rounds}], \forall p \in P \tag{7}$$

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

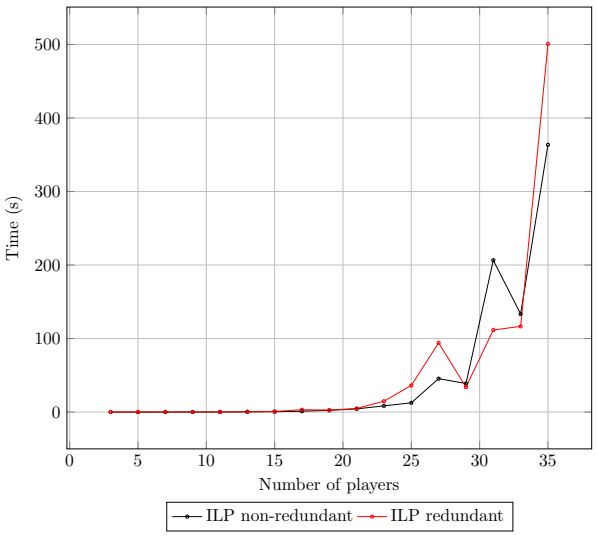

Figure 1: ILP runtime w.r.t the number of players

[0]We don't care about the elements in this set really, just its size. As we can't know if p plays as white or black we'll just care about the round, whatever.

3 Meta-heuristics

For the meta-heuristics, the pseudo-code of your constructive, local search and GRASP algorithms, including equations for describing the greedy cost function(s) and the RCL

Poner ecuaciones para el greedy cost y la RCL

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 construct 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.

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 p.points[day] (DESC)
6:     select p ∈ sortedPlayers[0]
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[1].

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.

[1] 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.

Algorithm 3 GRASP

```
1: Players ← Set of Players
2: rests ← {}
3: for day in 0..days do]b
4:     playersToRest ← filter Players(p) | p.hasNotRested
5:     sortedPlayers ← sort PlayersToRest(p) by p.points[day] (DESC)
6:     q_max ← sortedPlayers.first().points
7:     q_min ← sortedPlayers.last().points
8:     RCL_max ← {p ∈ sortedPlayers | p.Points >= q_max − α ∗ (q_max − q_min)}
9:     select p ∈ RCL at random
10:    rests[day] ← p
11: end for
```

We must repeat this algorithm iteratively, having some stop conditions as a MAX_ITER and a MAX_ITER_NOTIMPROVED. In this way, we put a maximum of iterations and a maximum of iterations without improve the objective function.

4 Parameter tuning

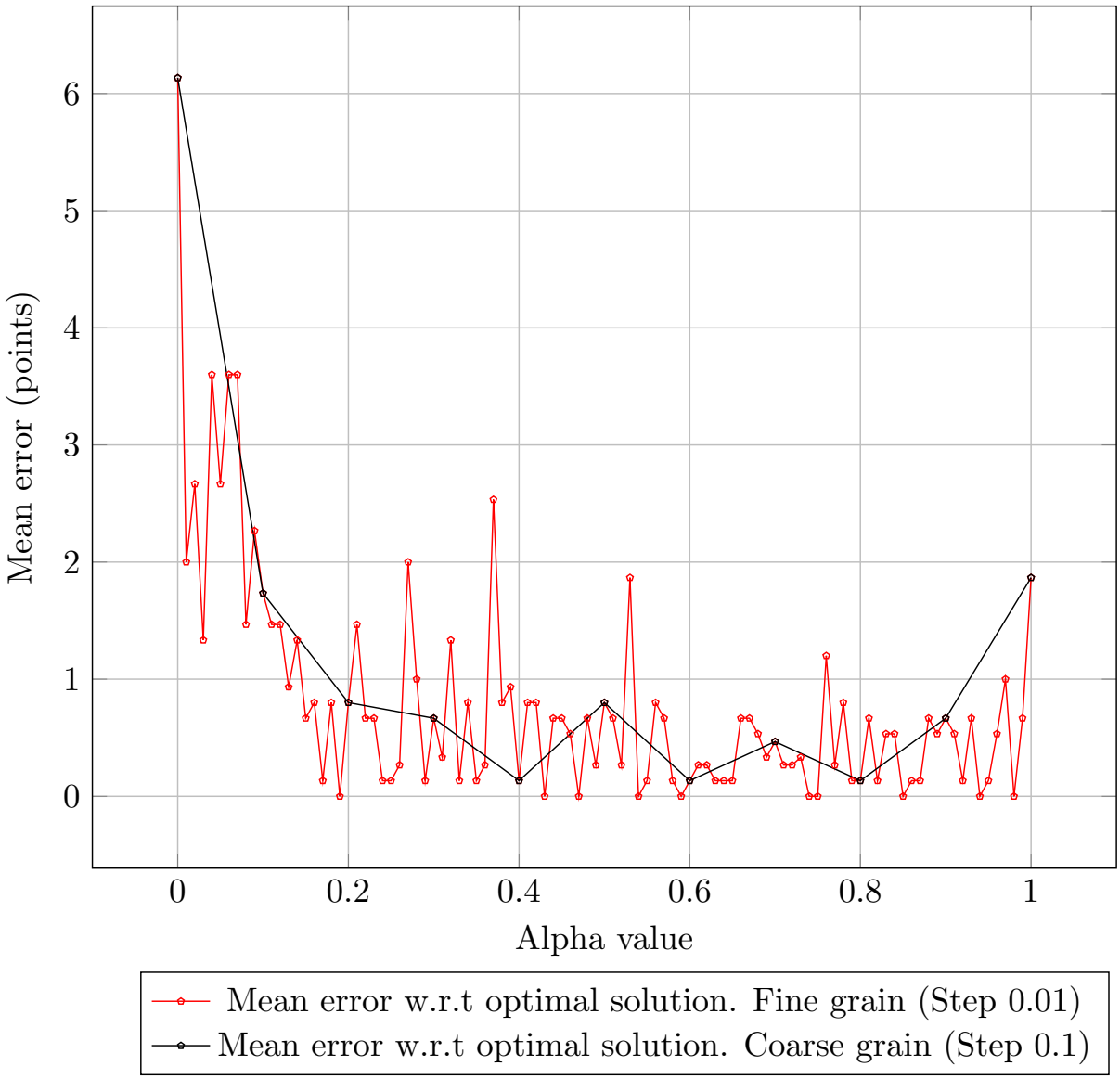

Figure 2: Mean error w.r.t alpha

5 Results

5.1 Time

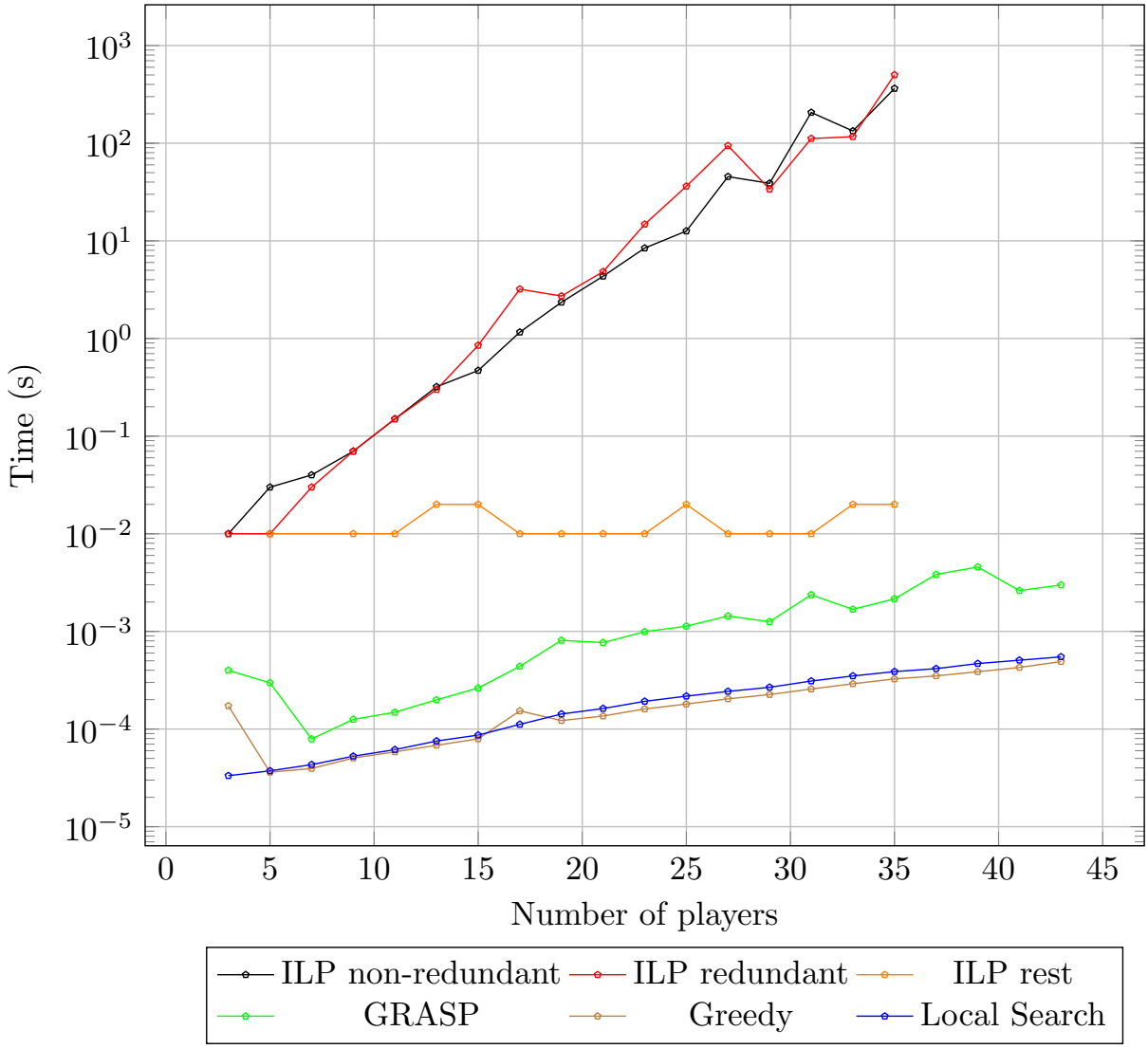

Figure 3:

5.2 Objective function

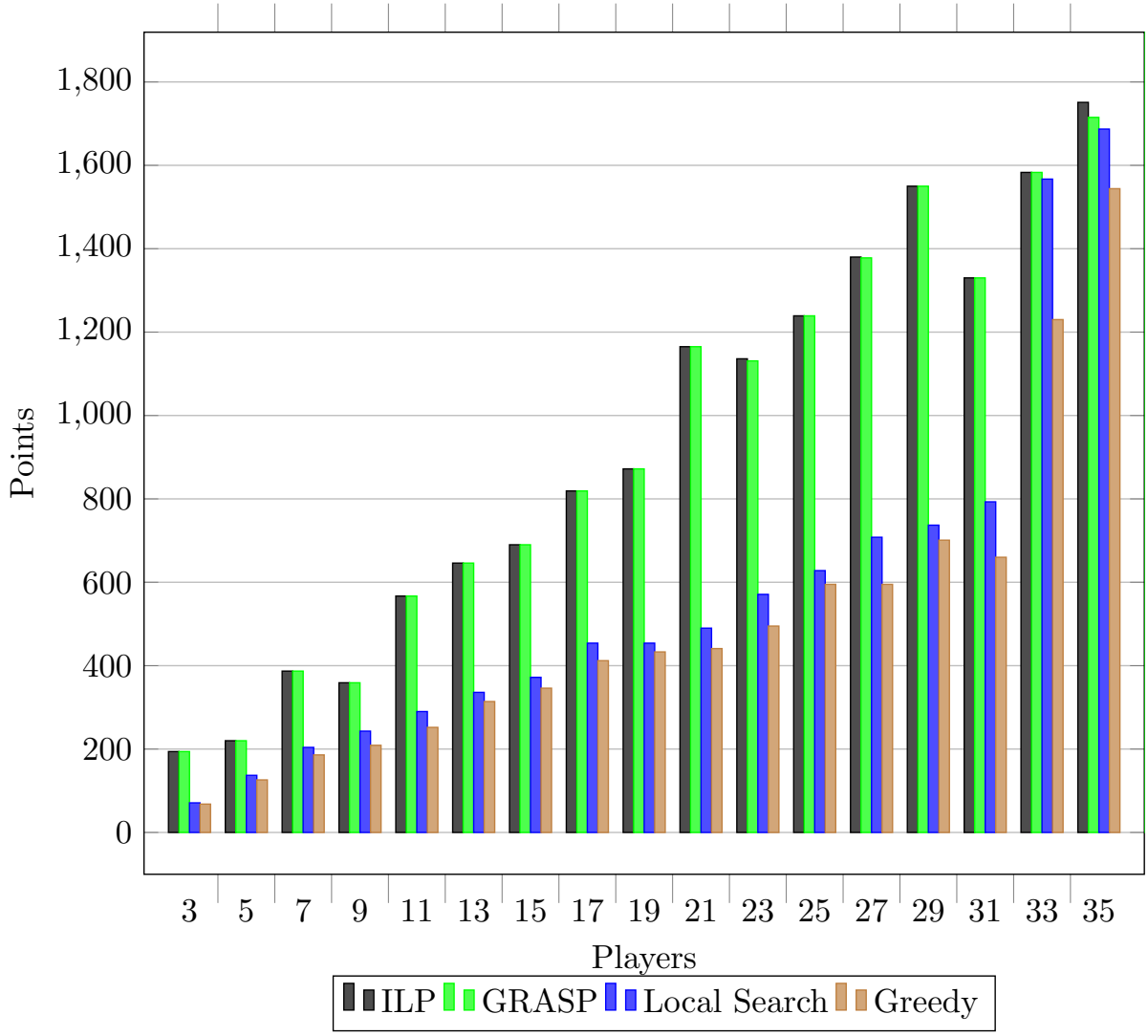

Figure 4: Objective function value per instance and solver

6 Reproducing the results

- OPL source code
- Programs of the meta-heuristics
- Instance generator
- Instructions how to use every of them and how to reproduce results