ALGORITHMIC METHODS FOR MATHEMATICAL MODELS

POLYTECHNIC UNIVERSITY OF CATALONIA

Course Project

Authors:

Mehdi Hassanpour
Keith Joseph D'souza

Fall 2022

1 Problem statement

In this problem we are scheduling a chess tournament for any odd number of participants. The tournament is held in several time slots each consisting of certain matches in parallel. Each player takes a slot off to "rest".
The problem can be formally stated as follows:

Given:

- Total number of players N
- Total number of slots N
- Matrix p of preferences. For each player i his/her preference to rest on slot j is specified as p_{ij}; Potentially adding p_{ij} to the scheduling score.

Find the best scheduling (assignment of matches to time slots) subject to the following constraints:

- Players cannot be matched against themselves.
- Each two participants must be matched during the tournament exactly once.
- In each slot, a player can play at most one match; If he/she is not resting exactly one game otherwise none. There are exactly $\frac{N-1}{2}$ games per slot.
- Each player takes exactly one slot off.
- On each time slot, there is exactly one player resting.
- During the whole tournament, a player must have an equal number of black and white games.

Objective is to *maximize* the scheduling score based on players' preferences on resting.

2 ILP Model

In this section we model the scheduling problem as a Integer Linear Program. Following sets and parameters are defined:

N	Set of players, index n
p	Square matrix of positive integer preference points for every player and time slot, index n, n for rows and columns

Now we define the decision variables:

x_{cs} Boolean variable showing if a match of two players is scheduled on a time slot $x_{cs}[i][j][k]$ is 1 if players i, j are matched on slot k.

$rest$ Boolean variable showing if a player is resting on a given slot or not. $rest[i][k]$ is 1 if player i is resting on slot k.

Finally the ILP model is defined as:

$$maximize \sum_{i \in N} \sum_{k \in N} rest[i][k] \times p[i][k] \tag{1}$$

$$\sum_{k} x_{cs}[i][i][k] == 0 \ \ \forall i, j \in N \tag{2}$$

$$\sum_{k \in N} (x_{cs}[i][j][k] + x_{cs}[j][i][k]) == 1 \ \ \forall i, j \in N, i \neq j \tag{3}$$

$$\sum_{i \in N} \sum_{j \in N} x_{cs}[i][j][k] == \frac{n-1}{2} \ \ \forall k \in N \tag{4}$$

$$\sum_{k \in N} rest[i][k] == 1 \ \ \forall i \in N \tag{5}$$

$$\sum_{i \in N} rest[i][k] == 1 \ \ \forall k \in N \tag{6}$$

$$\sum_{j,k \in N} x_{cs}[i][j][k] == \sum_{j,k \in N} x_{cs}[j][i][k] \ \ \forall i \in N \tag{7}$$

Implementation of the this model is provided in attached files.

3 Meta heuristics Approaches

Even though we managed to model the problem in CPLEX and solve it, we would like to investigate the performance of solutions by applying two heuristic methods **Greedy** and **Grasp**. Furthermore, we improve their solution by providing a **local search** algorithm. First let us discuss the problem one more time. Recall that in this tournament every contestant will play against every other one exactly once. Therefore, matches are predetermined and we only need to assign them to different time slots based on which player rests when. Therefore We claim that our optimization problem only relies on players preference on resting. Additionally, playing as black or white can be ignored for the optimization and managed during the scheduling. Because, we can change the colors accordingly. We break the main problem into three stages:

1. Given a set of rest priorities for each player (matrix p), find an order of resting for players (we will call this "rest vector" from now on) such that our scheduling score is maximized.
   After this stage, scheduling is unknown but we have achieved the highest possible score (based on algorithm) and the rest vector.

2. For an arbitrary rest vector, perform the scheduling for matches. Our approach here is to assume player i rests on slot i for simplicity; However, it does not matter and any other valid rest vector also works.

3. Map players based on the rest vector achieved in stage 1 and the assumption in stage 2. After mapping entire players, the final schedule in stage 2 matches exactly with the rest vector in stage 1.

Based on the implementation, stage 2 and 3 can be merged (mapped on fly).
Let us proceed with an example for a better understanding. Assume stage 1 has resulted in the rest vector $\{3, 4, 2, 1, 5\}$. And from stage 2 we have:
rest vector = $\{1, 2, 3, 4, 5\}$
slot 1:
player 1 is resting
matches: 2 vs 4, 3 vs 5
Now based on the two rest vectors we have:
$1 \rightarrow 3$
$2 \rightarrow 4$
$3 \rightarrow 2$
$4 \rightarrow 1$
$5 \rightarrow 5$
Therefore the matches on slot 1 will change to:
4 vs 1, 2 vs 5
As you might have noticed, now player 3 is not playing on the first slot and it is compatible with rest vector from stage 1.
Next we discuss our approaches for each individual algorithm in details.
Remember that the goal here is to optimize the scheduling score and as we mentioned earlier it only searches for the best rest vector. Assignment of matches to slots are performed separately and is the same for both algorithms.

3.1 Greedy

Consider the given matrix p regarding the points as stated in project description. IF there was no constraints, we could simply select the largest values of the matrix one by

one. However, this selection process is limited by two constraints:

$$p = \begin{bmatrix} 10 & 20 & 30 & 40 & 0 \\ 0 & 0 & 50 & 50 & 0 \\ 100 & 0 & 0 & 0 & 0 \\ 60 & 30 & 0 & 0 & 0 \\ 20 & 25 & 15 & 15 & 25 \end{bmatrix}$$

- At each slot, exactly one player rests. This is simply translated as for every column of the matrix, we should exactly select one value.
- Every player, rests exactly one day. Accordingly, for each row of the matrix only one value can be selected.

Now let's adapt our greedy algorithm to these constraints. At first, we search for the largest value in matrix, here it's 100 on row 3 and column 1 (player 3 rests on slot 1). Next we update the rest vector and add this value to our scheduling score. Finally the corresponding row and column are removed (subject to constraints above). We have:

$$p = \begin{bmatrix} 20 & 30 & 40 & 0 \\ 0 & 50 & 50 & 0 \\ & & & \\ 30 & 0 & 0 & 0 \\ 25 & 15 & 15 & 25 \end{bmatrix}$$

The next value to select is 50, since there are two we pick one randomly. Let's assume the one on third column.

$$p = \begin{bmatrix} 20 & 40 & 0 \\ & & \\ & & \\ 30 & 0 & 0 \\ 25 & 15 & 25 \end{bmatrix}$$

We repeat this process until p is empty which means our rest vector is complete.
Notice that since we are trying to maximize the score, the higher the selected value is, its cost is lower (if we consider "cost" a negative concept and try to minimize it). Here is the pseudocode for this algorithm:

3.2 GRASP

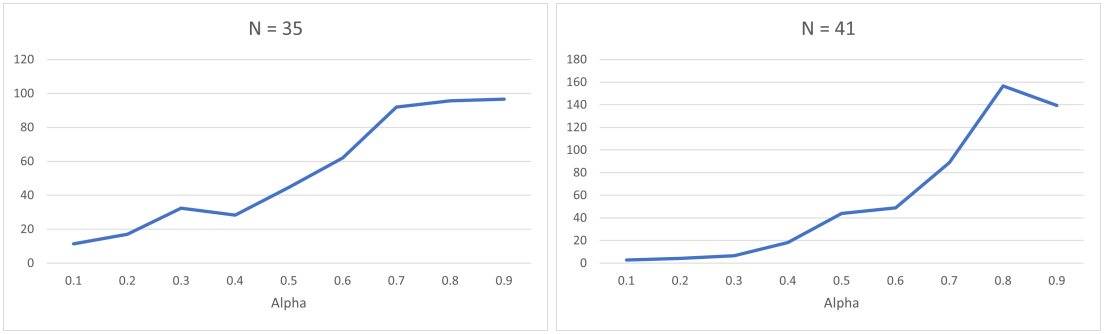
The general approach in GRASP is similar to the one explained in greedy however, here instead of selecting one value at a time and removing the corresponding values from matrix, we select a set of potential candidates and based on the parameter α we create our RCL list. Finally, one element of the RCL is randomly selected.

Algorithm 1 Constructive greedy algorithm

$p \leftarrow$ matrix of points
visited row $\leftarrow$ if player is already assigned
visited column $\leftarrow$ if slot is already assigned
while p is not empty **do**
    if i in visited row **then**
        continue
    end if
    if j in visited column **then**
        continue
    end if
    $V \leftarrow \max(p) : p_{ij}$
    visited row $\leftarrow$ visited row $\cup$ row i
    visited column $\leftarrow$ visited column $\cup$ column j
end while

Algorithm 2 Constructive GRASP algorithm

$p \leftarrow$ matrix of points
$S \leftarrow \{\}$
initialize C
while S is not a solution **do**
    $q_{\min} \leftarrow \min\{p_{ij} | p_{ij} \in C\}$
    $q_{\max} \leftarrow \max\{p_{ij} | p_{ij} \in C\}$
    $TH \leftarrow q_{\max} - \alpha(q_{\max} - q_{\min})$
    $RCL \leftarrow \{c \in C | p_{ij}\} \geq TH$
    select c randomly from RCL
    $S \leftarrow S \cup c$
    Update C
end while



(a) (b)

Figure 1: Quality of GRASP solution compared to CPLEX

Since we keep track of assigned slots and players the procedure for selecting candidates is very straightforward. At each iteration while adding all possible values of matrix p to the candidate list, we update the values for $q_{\min}$ and $q_{\max}$. The threshold TH is the parameter that controls the RCL set. Notice that for the first iteration, we manually change α to zero since we want to select the first element of the solution as greedy. In our implementation these candidates are tuples of player and slot index. Therefore, their quality is simply achieved as p_{ij}.

3.2.1 Alpha

In order to find the best value of alpha for our experiments, we generated two instances of different size (35, 41) and ran the constructive phase of GRASP algorithm three times for different values of alpha. Figure **??** and **??** show the quality of average solution compared to the results from CPLEX. As we can see for $\alpha = 0.1$ GRASP provides a better quality solutions.

α	35			Quality	CPLEX	41			Quality	CPLEX
0.1	307	310	303	11.333	318	348	348	346	2.667	350
0.2	295	311	297	17	318	349	341	347	4.333	350
0.3	284	282	291	32.333	318	350	341	339	6.667	350
0.4	276	300	293	28.333	318	348	323	324	18.333	350
0.5	278	285	257	44.667	318	320	315	283	44	350
0.6	258	264	246	62	318	321	291	291	49	350
0.7	235	214	229	92	318	257	251	275	89	350
0.8	176	231	260	95.667	318	182	204	194	156.667	350
0.9	241	176	247	96.667	318	260	195	177	139.333	350

Table 1: Tuning Alpha and comparing GRASP results with CPLEX

3.3 Local Search

As we already learned from the lectures, sometimes these heuristic algorithms are stuck around a local minimum (maximum) and therefore, may not achieve the best results. However, using additional techniques such as local search will potentially improve the solutions. Once again, this algorithm is only applied on the rest vector.
Our local search algorithm for a given solution examines further solutions by swapping every two element of the rest vector. Based on their values in matrix p the cost is calculated and the swap is finalized if a higher score is achieved.

Algorithm 3 Local search algorithm

1: $p \leftarrow$ matrix of points
2: $i \leftarrow 0$ to N
3: $j \leftarrow i+1$ to N
4: $S \leftarrow$ rest vector from greedy or grasp
5: $S' \leftarrow$ swap S_i with S_j
6: **if** $score(S') > score(S)$ **then**
7: $\quad S \leftarrow S'$
8: **end if**

When using local search, it can be seen that our algorithm actually improves the solutions in some cases.

4 Results

After generating instances of various size, we solved the problem with each algorithm. Figures 2, 3 and tables **??**, **??** show these results. There is a significant difference between execution times of heuristic algorithms and CPLEX however, with the help of local search quality of solutions is acceptable.

	Execution Time				
Size	CPLEX	Greedy	Greedy + Local search	GRASP	GRASP + Local search
21	5.38	0.001891	0.0130255	0.010999	0.0079964
23	5.3	0.008936	0.0161211	0.010909	0.0139872
29	30.39	0.011648	0.0120046	0.007998	0.025685
31	221.66	0.006915	0.0111835	0.008015	0.011914
35	408.97	0.011218	0.00691	0.013912	0.0331482
41	1274.13	0.011022	0.0101421	0.016004	0.0150576
43	> 2880.00	0.012021	0.0305429	0.019381	0.032996

Table 2: Comparison of execution time with different algorithms and different sizes

	Objective Value				
Size	CPLEX	Greedy	Greedy + Local search	GRASP	GRASP + Local search
21	315	302	311	308	315
23	342	337	340	337	340
29	347	342	343	333	340
31	376	367	368	364	369
35	318	318	318	311	318
41	350	350	350	346	350
43	396	386	388	384	391

Table 3: Comparison of objective value with different algorithms and different sizes

5 Using the program

Here we explain the structure and requirements to execute out program.

5.1 Requirements

Greedy, GRASP, local search and instance generator are implemented in C++11. Some basic features of version 11 allowed us to interpret the input as regex and perform the execution times using *chrono* library. All the libraries and functions used in this project can be found in the standard libraries of C++.

5.2 Configuration

Alongside the project, there is a file named *config.h* controls the parameters such as input/output file names, instance names, instance size, enabling/disabling local search, the method (greedy/grasp) and Alpha. Notice that parameters *instance_file_name*, *instance_size* and *number_of_instances* are the parameters used only by the instance generator. Edit them to get instances with different sizes or multiples of the same size.

5.3 InstanceGenerator

This program (*instanceGenerator.cpp*) creates a set of random inputs for the problem based on the given size. Based on our observation on the generated numbers, we realized simply using the *rand* function does not really provide us with a uniformly distributed set of numbers. In other words, for instance sizes of greater than 10, only the first elements of each row were non-zero. To generate the random numbers of the points matrix, there are two different algorithms implemented.

- Generate a random number R between 0-100 initially. Subtract it from 100 and generate the next one between 0-R.

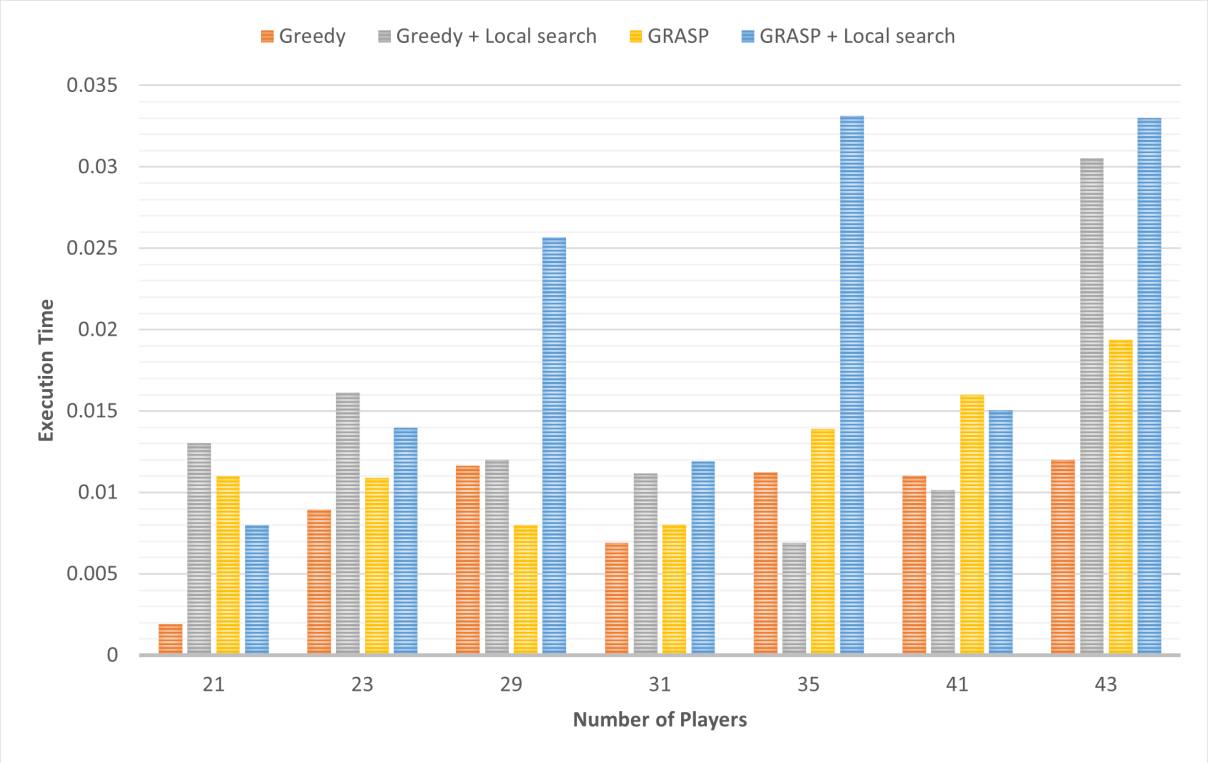


Figure 2: Comparison of execution time with different algorithms and different sizes

- Generate a set of $N + 1$ random numbers between 0 and 100. Subtract each two successive elements and add it to the matrix.

Since both of these algorithms together can results in a more distributed instances, we implemented both and each time we choose one randomly during execution.

5.4 Run

After configuring *"config.h"* running the following commands should provide you the programs. This step is required every time we change the configuration:

```
g++ -std=c++11 main.cpp -o solve
```

To use instance generator:

```
g++ -std=c++11 instanceGenerator.cpp -o instanceGenerator
```

You should be able to run the two execution files *instanceGenerator* and *solve* afterwards.

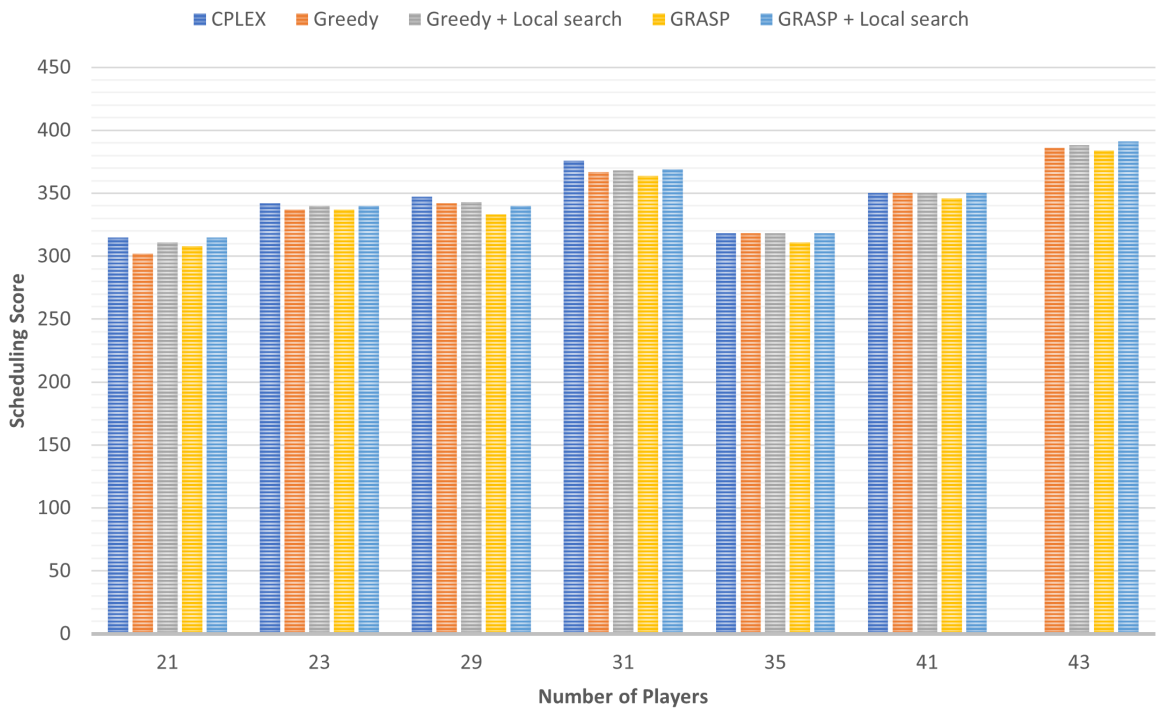


Figure 3: Comparison of objective value with different algorithms and different sizes

The main file of in *main.cpp* uses a function named *solve* in which based on the greedy or grasp methods, calls the corresponding procedures.

Students' contributions

Mehdi Hassanpour and Keith Joseph D'souza have worked together in order to understand the problem find the solutions. Mehdi has been in charge of implementations of ILP Model, Greedy, GRASP, Local search and instance generator.