# Practice test on Game Theory

Ehsan Shaghaei 09.12.1999

February 13, 2022

## 1 Code Structure

### 1.1 FPG class

FPG class, ipso facto *Finite Position Game*, partially contains the parameters of a formal FPG. what makes this class special is exploring and extracting winning positions with a well defined strategy aka backward induction.Rest of the class contains class attribute initialization. The backward-induction implementation is as following :
Firstly, we assume that we "lead the opponent to loose " resembling that we won the game by moving to **the final position** of the game[line 25]. Let's suppose our game scenario is consist of a board with corresponding consecutive positions on the FPG and players in turns move a marble from a facing[initial] *"position on board"* and whoever starts moving his marble with *"position on board"* equal to the game's *final position* is another resemblance of loosing the game for that player.To clear up, let us define the **"position on board"** as the position a player faces on the board before making his move. As we are interested to win/lead the opponent to loose; we are interested to make the aforementioned loose position the **"position on board for the opponent player"**.

```
1  class FPG:
2      def __init__(self, dd: int, mm: int, yyyy: int):
3          # deleted lines....
4          # The game initiatives according to the lecture and the examination document
5          self.positions = SortedSet(range(1, dd+mm+yyyy+1))
6          self.moves = SortedSet(range(1, dd+mm+1))
7          self.finalPosition = dd+mm+yyyy
8          # Initiallizing Variables feild
9          # to store positions which can
10         # positions which will lead to win of the player against us
11         self.winningPositions = SortedSet()
12         # positions which will lead to loose of the player against us
13         self.losingPositions = SortedSet()
14         # moves which will lead to our win
15         self.winningMoves = SortedDict()
16         # Just the method to fill the feilds above using back induction
17         self._solveGamebyBackInduction(positionOnBoard=self.finalPosition)
18 # receives the position on the board of the game before the player makes any move and thus exploits the strategy.
19     def _solveGamebyBackInduction(self, positionOnBoard):
20         # first of all we check if the position we are using is a admissable position in our FPG
21         if self.positions.__contains__(positionOnBoard):
22             # if the player X wants to make a move and look at board and the other player already landed on the
       final position, thus the player X lost.
23             # let us call it a loosing position
24             if positionOnBoard == self.finalPosition:
25                 self.losingPositions.add(self.finalPosition)
26             # now lets iterate over the moves which helped us to land on a positions to make the player against us
       loose.
27             for move in self.moves:
28                 # determines if we are not checking invalid moves. specialling if we are at position 2 and check if
       a hop of 21 brang us here :D avoid contradiction
29                 if self.positions.__contains__(positionOnBoard-move):
30                     # This is a memorable position:D, it helped us to win. so let's store it as a position which
       will lead us to win
31                     self.winningPositions.add(positionOnBoard-move)
32                     # Lets not forget where we came from, this was a move which helped us to make our Opponent loose
33                     self.winningMoves.setdefault(
34                         positionOnBoard-move, positionOnBoard)
35
36         # well we checked only for one move, lets do the back tracking again, in otherwords let's do back-induction.
37         # First lets make sure the room we gotta back track in valid positions
38         if self.positions.__contains__(positionOnBoard-max(self.moves)-1):
```

```
39              # Now we know since from this position we can not lead to a loosing point for our openent, thus it is
    another loosing point so lets mark it as a loosing point.
40              self.losingPositions.add(positionOnBoard-max(self.moves)-1)
41              #  Now, Let's backtrack which moves will lead to this position.
42              self._solveGamebyBackInduction(positionOnBoard-max(self.moves)-1)
```

Listing 1: FPG solveGamebyBackInduction method

Obviously,we must first make sure that our target "position on board" is a valid position[lines 21,29 and 38] so that we know we won't place our marble outside the game board.Then we iterate over all the admissible moves which helped us to lead the opponent to target loosing position;[lines 27-34] the source of these moves can be labeled as a winning *"position on board"*[line 31] for any player, since that player can move his marble from the winning on board position with admissible moves to a loosing on board position for the opponent and store this move as a winning move[line 33]. Moreover, **one position before** the *winning position* which can make*the longest admissible winning move* is again a loosing position; since we can not make any admissible move so that it will lead to **a loosing on board position**; thus the position itself is a *loosing on board position*[Line 27] and similarly, we trace back *the winning moves* and *winning position-on-boards* which can lead our opponent to a the new found losing on-board-position[Line 29] applying the same logic. The aforementioned explanation is backbone of my winning strategy proof in the theoretical part.please refer to my theory test for formal explanation and proof.

## 1.2 AdvisorBot class

This class is intended to give a possible advice for a move in a particular position with a predefined winning strategy on a FPG class instance and it's mode which can be either **RANDOM** or **SMART**[line 4-6].

```
1  # an advisor bot which exploits the winning positions regarding it's FPG game and mode.
2  class AdvisorBot:
3      # initialized the Advisor object by a @object FPG and game mode
4      def __init__(self, fpg: FPG, mode: GameMode):
5          self.fpg = fpg
6          self.mode = mode
7      # makes an advice with a possible winning strategy on a given position
8      def makeAdvice(self, position):
9          # checks if the given position is a valid position
10         if not self.fpg.positions.__contains__(position):
11             raise Exception("Invalid Position "+str(position))
12         # giving an advice regarding the given strategy [smart]
13         if (self.mode == GameMode.SMART) and (not self.fpg.losingPositions.__contains__(position)):
14             return self.fpg.winningMoves.get(position)
15         # giving an advice regarding the given strategy [random]
16         elif (self.mode == GameMode.RANDOM):
17             return min(random.choice(self.fpg.moves)+position,myfpg.finalPosition)
18         # chooses random strategie if anything went wrong.
19         else:
20             return min(random.choice(self.fpg.moves)+position,myfpg.finalPosition)
```

Listing 2: AdvisorBot class

This class features a *makeAdvice* method which firstly evaluates if the position is a valid position in the FPG game it knows[Lines 10-11].In case the *AdvisorBot* is set on a *RANDOM* mode, it tries to make a random choice in admissible moves in it's given FPG from the given position, and in case choice of a move would lead to an invalid position, it will choose the closet valid move[Lines 16-18].In case the *AdvisorBot* is set on a *SMART* mode, firstly it checks if the given position is not a *Loosing-On-Board position*[defined in part 1.1][Line 13] and then returns the destination of the corresponding winning move from it's known FPG; Additionall if the given position was a *Loosing-On-Board position* it makes a random choice in admissible moves in it's given FPG, and in case choice of a move would lead to an invalid position, it will choose the closet valid move. [Line 19-20]

## 1.3 Miscellaneous Parts

The rest of source code does not have much of a technical value thus I prefer abstinence of detailed explanation. Briefly speaking, classes and enums *TextConst,LogConst,GameMode* were used for avoiding hard-coding and improving code re-usability. The required log file is a .json file called *'game.json'* which will automatically be saved in the source directory after the game. The log file contains: the game mode, the moves and corresponding player who made the move, and the game winner.The program is not meant to handle user bad inputs, however it tolerates some basic user mistakes. please avoid misusing the program and pay attention to the prompt.