Andrey Vagin
12.09.2002

BS20-AI
25th of September

The First Practical Test

Problem:

Implement functionality of Finite Position Game between Duplicator and Spoiler as given in the task description.

Intoduction:

The finite position game can be defined as a tuple $G = (P_A, P_B, M_A, M_B, F_A, F_B)$ as presented in the Lecture 1, slide 15. Taking into account the description of FPG from the task description and applying my date of birth, the sets will be as follows:

- $P_D = \{(p, D) : D = 1 \land p \in \land p \geq 1 \land p \leq 2023\}$
- $P_S = \{(p, S) : D = 2 \land p \in \land p \geq 1 \land p \leq 2023\}$
- $M_D = \{((p, D), (q, S)) : D = 1 \land S = 2 \land p \in P_D \land q \in P_S \land (\exists n)[n \in \land n \geq 1 \land n \leq 21 \land q = p + n]\}$
- $M_S = \{((q, S), (p, D)) : D = 1 \land S = 2 \land p \in P_D \land q \in P_S \land (\exists n)[n \in \land n \geq 1 \land n \leq 21 \land p = q + n]\}$
- $F_D = \{(2023, D) : D = 1\}$
- $F_S = \{(2023, S) : S = 2\}$

In my implementation, I excluded the final position from the set of initial positions in order to eliminate the uncertainty that arises when starting from such a position.

The technique called "backward induction" stands for the process of iterative inspection whether the position has a winning strategy. According to Lecture 1, slide 20, there can be defined function $\mathcal{F}(W) = [2002..2022] \cup (\{p : (p+1) \notin [2002..2022], (p+2), (p+3), ..., (p+22) \in W\}) \cup (\{p : (p+2) \notin [2002..2022], (p+3), (p+4), ..., (p+23) \in W\}) \cup ... \cup (\{p : (p+21) \notin [2002..2022], (p+22), (p+23), ..., (p+42) \in W\})$ on $[1..2023]$ representing all the positions with a winning strategy. Formally, it involves the Tarski-Knaster Theorem to define set of fix-points for the function $\mathcal{F}(W)$.

Manual

The framework is implemented in the Python programming language of version 3.8.8. In order to run the program run use:

```
$ python3 AndreyVagin.py
```

On setup program will create a directory called 'log_files', where all the logs will be stored. This directory will also include 'datafile.json' where number of played sessions will be stored. Log file starts with a configuration chosen by user, and follows with the sequence of moves for both players. After end of each play log file is saved with new index equal to the number of saved plays + 1.

The session starts with a game analysis, i.e. the program computes fix points for given conditions. Then program asks to configure the game: it prompts user to choose whether the starting position should be selected randomly or manually, and decide on the gaming mode (either smart, random, or advisor). Further players start make moves. Players move in turns Duplicator-Spoiler-Duplicator-Spoiler-etc where the user is Duplicator. The game ends when the final position is reached. At the end user could choose to keep the session and play again.

Framework designed to be tolerable to user mistakes. For instance, if the format of answer is not expected, the program will ask to answer again. The context messages guide user throughout the game session. Another important note is handling keyboard interrupt: user can stop the game whenever he/she wants.

Design

One of the goals in the development was designing an architecture that guarantees re-usability for solving similar FPG problems. Method which implements backward induction called *find_lose_points()* is parameterized and invoked with needed game's conditions. It returns a set of fix point for a given game's setup. The logic of the method is an implementation of formula provided in the **Introduction** section. This formula is applied iteratively as in the lecture slide 30.

There are methods which are implemented for tolerant reading of user's input for configurations and moves. These methods are: *choose_init_postion()*, *choose_playing_mode()*, *read_user_input()*.

smart_comp_move() applies wining strategy for the computer if it is possible. If computer can it moves to the nearest point that is not the set of fix points, after such move the user will not be able to win the computer. If computer can not make described move it makes random move.

advice_move() method is used to suggest the user the wining strategy if it is possible. Suggested strategy is the same that is used for *smart_comp_move()*. If there is no wining strategy for the given point method prints about it.

game() represents a flow of a game. It stores current score between Spoiler and Duplicator and iteratively calls *play()* method that implements the flow of each play.