# Practice test 1: Finite Position Game

Danis Alukaev (12.11.2001)
d.alukaev@innopolis.university
B19-DS-01

February 13, 2022

**Abstract**

The paper presents yet another implementation of Finite Position Game (FPG) provided in task specification. Significant efforts were directed to design easy-to-reuse python framework implementing backward induction for FPG's. This document follows the report structure.

# Introduction

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 theory test and applying my date of birth, the sets will be as follows:

- $P_D = \{(p, D) : D = 1 \wedge p \in \mathbb{Z} \wedge p \geq 1 \wedge p \leq 2024\}$
- $P_S = \{(p, S) : D = 2 \wedge p \in \mathbb{Z} \wedge p \geq 1 \wedge p \leq 2024\}$
- $M_D = \{((p, D), (q, S)) : D = 1 \wedge S = 2 \wedge p \in P_D \wedge q \in P_S \wedge (\exists n)[n \in \mathbb{Z} \wedge n \geq 1 \wedge n \leq 23 \wedge q = p + n]\}$
- $M_S = \{((q, S), (p, D)) : D = 1 \wedge S = 2 \wedge p \in P_D \wedge q \in P_S \wedge (\exists n)[n \in \mathbb{Z} \wedge n \geq 1 \wedge n \leq 23 \wedge p = q + n]\}$
- $F_D = \{(2024, D) : D = 1\}$
- $F_S = \{(2024, S) : S = 2\}$

In the program there is possibility to start the game in the final position, so the possible starting positions are not limited and are $P_D$ and $P_S$. Although, there is an ambiguity in interpretation of the outcome: according to different sources it will lead either to win or lose. My program treats the start in the final position as a win of the Duplicator.

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 $\mathcal{F}(W) = [2001..2023] \cup (\{p : (p+1) \notin [2001..2023], (p+2), (p+3), ..., (p+24) \in W\}) \cup (\{p : (p+2) \notin [2001..2023], (p+3), (p+4), ..., (p+25) \in W\}) \cup ... \cup (\{p : (p+23) \notin [2001..2023], (p+24), (p+25), ..., (p+46) \in W\})$ 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 $\mathcal{F}(W)$. Implementation of "backward induction" logic will be fundamental in the Design section.

# 1 Manual

The framework is implemented in the Python programming language of version 3.8.12. In order to run the program, navigate to the directory with downloaded from Moodle script. Make sure that you have permissions to modify the directory. Further, activate the environment with appropriate version of Python. Run the script using

```
$ python3 DanisAlukaev.py [-h] [-b BIRTHDAY]
```

By default for all computations the date is set to be my birthday. Still, you can specify custom date in Command Line Interface (CLI). The documentation on developed CLI available via "–help" flag.

On setup program will create a directory called "logs", where all the logs will be stored. There are two levels of logging. The first one is standard output logging that captures all the data sent to stdout in terminal. This information can be used for developers to reproduce user's actions for debugging. The second level is saving the records about each play independently. This file starts with a configuration chosen by user, and follows with the sequence of moves for both players. One session might have several plays, so the name of log file is a conjunction of session datetime along with serial number of play.

The session (see Figure 1) starts with a game analysis, i.e. the program computes set of positions that have winning strategies. On average it takes 10 seconds. Once the job is done, the 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 (the user is always first). The game ends when the final position is reached. At the end user could choose to keep the session and play again (may be with different configuration).

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.

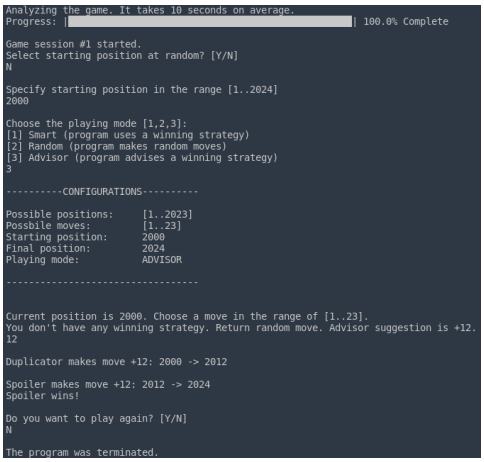


Figure 1: Example of the play session

# 2 Design

One of the major goals in the development was designing an architecture that obeys OOP-design standards and guarantees reusability for solving similar FPG problems. In the course of work, there were introduced four key abstractions constituting FPG: *Analyzer*, *Configuration*, *Spoiler*, *Game*.

*Configuration* is a base class to create FPGs. It contains the prompt routines to configure the starting position and playing mode. Its also set-ups custom logger class. This class redirects the standard input and output to the log file (see 1, first level logging). The motivation behind this class justified by the complexity of game process. Indeed, it becomes quite handy to split up all the auxiliary methods from the main functionality.

*Game* inherits from the class *Configuration*. This class is in charge of interactive game sessions that follow the formulation of FPG given previously. It also aggregates classes *Analyzer* and *Spoiler* representing the game environment. The *Analyzer* is used for advisor mode, and suggests the optimal moves for the user. Besides, the second level logging (see 1) is implemented in this class.

*Analyzer* implements the "backward induction" described in the Introduction, i.e. it iteratively checks whether there exists a possible move that does not lead to the final position, and any opponent's move leads to position with a winning strategy. During the game it acts as consultant: for user in "advisor" mode, and for Spoiler in "smart" mode. This performed via a method that returns the optimal move for a given position. The list of optimal moves for each position is generated during "backward induction". If current position does not have winning strategy, the program returns random move among all possible and notifies the user.

*Spoiler* implements opponent's behaviour. Depending on playing mode either chooses the optimal move (smart/advisor) or makes moves at random (random mode).

# 3 Limitations

The "backward induction" routine in *Analyzer* for a given task performs for 10 seconds on average. The performance is expected for a chosen method, but might improved by replacing the technique by any other. If the generalization is not the aim for the task, the winning strategies could be find in linear time: there can be provided a mathematical prove that each $day + month + 1$ integer (since the end) will not have a winning strategy.