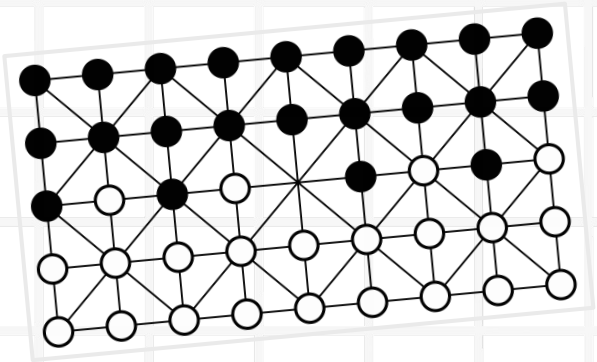
# Fanoron-Tsivy

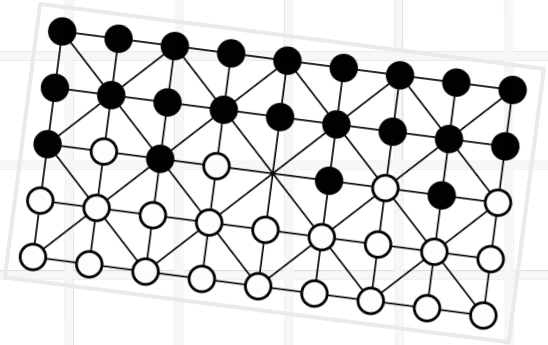


**T11G04 | Artificial Intelligence**

Luís Duarte (202108734) | Madalena Ye (202108795) | João Figueiredo (202108873)

# Game Definition

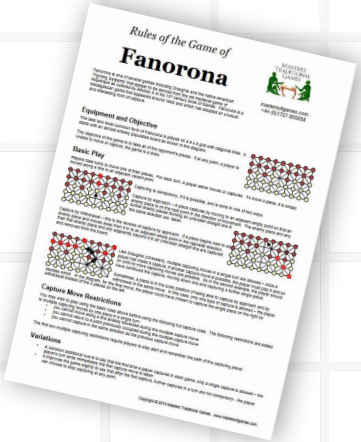
- Fanoron-Tsivy or Fanorona is a **two-player strategy game**. The best and most common form of Fanorona has a board which consists of lines and intersections, creating a grid with **5 rows** and **9 columns** subdivided diagonally.
- Players take **turns** moving pieces on a board, with the **goal of capturing all of the opponent's pieces**.
- Capturing is **compulsory** and can be executed in two distinct manners: **moving a piece to an empty adjacent point** where an enemy piece lies in the direction of movement, or **moving away from an adjacent** enemy piece into an empty point in the opposite direction.
- There can be **multiple capturing moves** in a single turn, mandating players to continue capturing if possible until no further captures are feasible, thus emphasizing the **importance of foresight and planning** in gameplay.

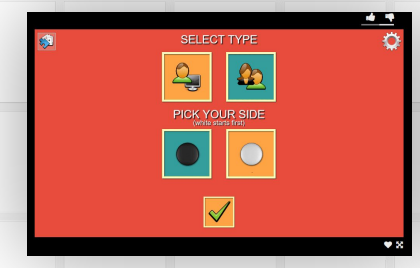


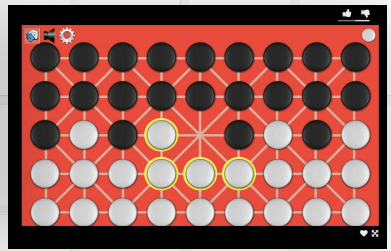
# Related Works & References

- To learn and understand the rules of the game we found a very clarifying PDF by "Masters Traditional games" that explained the nuances of the game.

- To get familiar with the rules of the game we engaged in various matches against an already implemented AI bot we found online

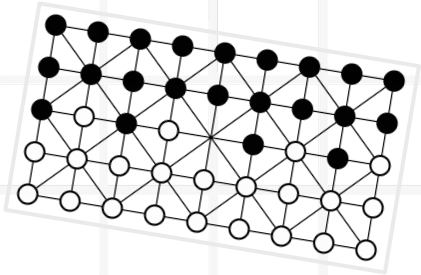

Rules of the Game of
Fanorona



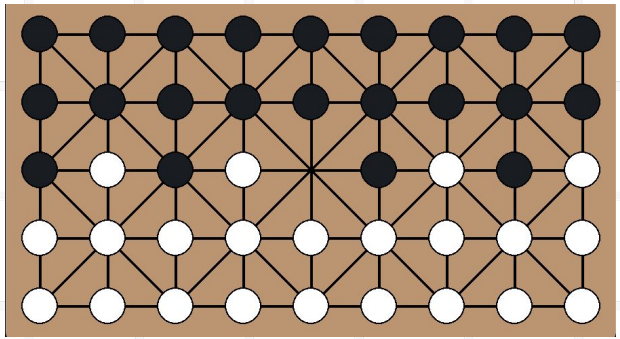
# State Representation

- Initial state:  A tuple that consists of: the initial board and the initial player
- Operators (in the case of an attack, if the player can attack another line it can repeat the attack operator):
  - Attack by approach
  - Attack by withdrawal
  - Move
- Objective test: Checks if enemy doesn't have any remaining pieces
- Path cost: Number of enemy pieces at certain state (this will incentivize a search algorithm to try and take the most pieces)



# Implementation Work

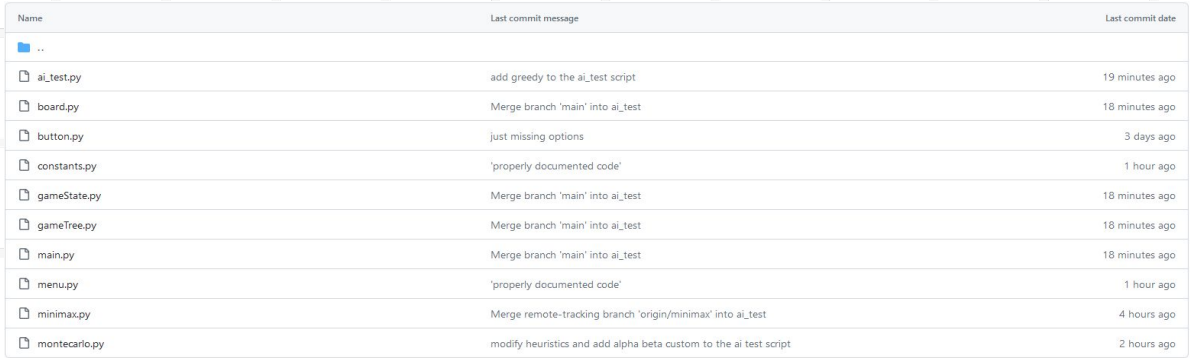
- Written in Python and leverages PyGame to draw the game
- Game state is fully implemented (move generation, move execution)
  - Not trivial because we cannot repeat the same “line” therefore, it makes move generation harder and slower.
- Game is playable through the GUI (which supports PvP, PvM and MvM)



Our board

# Data Structures & Other Implementation Details

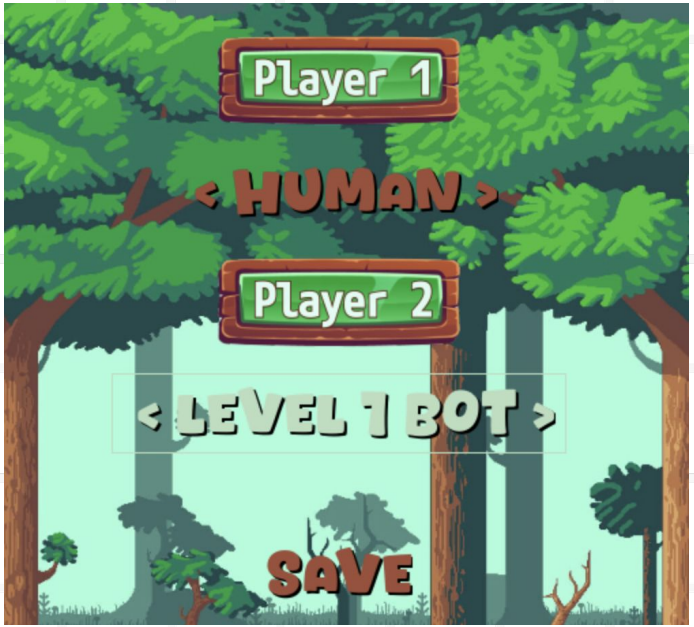
- **Game State:** Responsible for managing the **game's logic**, our game state module encapsulates data pertaining to the **current arrangement of pieces**, **evaluates available moves**.
- **Board:** Integral to the **graphical representation** of the game, the board class module oversees the **visual rendering of the game board** and the distinctive **pieces belonging to each player**.



| Name | Last commit message | Last commit date |
|---|---|---|
| .. | | |
| ai_test.py | add greedy to the ai_test script | 19 minutes ago |
| board.py | Merge branch 'main' into ai_test | 18 minutes ago |
| button.py | just missing options | 3 days ago |
| constants.py | 'properly documented code' | 1 hour ago |
| gameState.py | Merge branch 'main' into ai_test | 18 minutes ago |
| gameTree.py | Merge branch 'main' into ai_test | 18 minutes ago |
| main.py | Merge branch 'main' into ai_test | 18 minutes ago |
| menu.py | 'properly documented code' | 1 hour ago |
| minimax.py | Merge remote-tracking branch 'origin/minimax' into ai_test | 4 hours ago |
| montecarlo.py | modify heuristics and add alpha beta custom to the ai test script | 2 hours ago |

# GUI


FANORONA
Start
Options
Quit


Player 1
< HUMAN >
Player 2
< LEVEL 1 BOT >
SAVE

# The Approach

- **Evaluation function:** We have devised two distinct game state evaluation functions integral to our algorithms. These functions play a pivotal role in assessing the board state and guiding subsequent decisions. The first one simply calculates the difference in the number of pieces of each player, while the second one incorporates a nuanced consideration of the strategic significance of each piece's placement according to the current board configuration.
- **Operators:** We introduced the ENUM Capture Type to categorize maneuvers as either 'APPROACH' or 'WITHDRAWAL', or to denote absence of capture. With this feature, our system efficiently checks move validity, simulates scenarios, and executes actions with precision, enhancing overall performance.

# Implemented Algorithms

- In total, we have developed **three** distinct algorithms along with their variations as part of our implementation process. In total we provide the user with **six different difficulty** levels
- **Level 1- *Greedy algorithm***, using the simple version of the game state evaluation function, in other words, it prioritizes immediate moves based on optimizing the piece ratio relative to the opponent.
- **Level 2- *MonteCarlo algorithm*,** it's a pure MCTS implementation based on random moves only slowed by Python itself.
- **Level 3- *MonteCarlo algorithm (Custom)*,** it implements a custom simulation policy where it only selects the moves where have the maximum number of pieces taken, if any. If there aren't any captures available it tires to reduce the "spread" of it's pieces to try and try to stop late-game "throws".

# Implemented Algorithms

- **Level 4-** ***Minimax algorithm (Depth 3)***, for the fourth difficulty level we implemented the standard version of the minimax algorithm with no cuts, employing the simple evaluation function, this implementation aims to select the optimal move that, within three iterations, leads to the most favorable piece ratio outcome.
- **Level 5-** ***Minimax A/β algorithm (Depth 5)***, building upon our previous implementation, while incorporating alpha-beta pruning. This enhancement significantly boosts performance by minimizing the number of nodes explored during the decision-making process, allowing for a larger depth.
- **Level 6-** ***Minimax A/β algorithm (Depth 5, heuristic variaton)***, similar to the level 5, the only difference is that the algorithm calls upon a different evaluation function, the more complex one that takes into account the strategic value of positions.

# Experimental Results

| AI 1 | AI 2 | W | L | D |
|---|---|---|---|---|
| Greedy | Minimax | 0 | 15 | 15 |
| Greedy | Montecarlo | 24 | 3 | 3 |
| Custom Montecarlo | Minimax | 0 | 30 | 0 |
| Minimax | Montecarlo | 30 | 0 | 0 |
| Custom Montecarlo | Greedy | 2 | 28 | 0 |
| Alpha-Beta | Minimax | 15 | 0 | 15 |
| Alpha-Beta Custom | Montecarlo | 30 | 0 | 0 |
| Alpha-Beta | Alpha-Beta Custom | 15 | 15 | 0 |
| Alpha-Beta | Greedy | 30 | 0 | 0 |
| Alpha-Beta | Custom Montecarlo | 30 | 0 | 0 |
| Alpha-Beta Custom | Minimax | 15 | 0 | 15 |
| Alpha-Beta Custom | Custom Montecarlo | 30 | 0 | 0 |
| Custom Montecarlo | Montecarlo | 15 | 15 | 0 |
| Alpha-Beta Custom | Greedy | 30 | 0 | 0 |

All algorithms run in < 5 seconds per turn

30 games - 15 per side;

>100 moves = draw;

MCTS capped to 5 seconds

# Analysis and Conclusions

- Increasing the depth of the minimax will make the AI stronger, however there's diminishing returns for this;
- The Monte Carlo Tree Search results are contradictory, because it's really strong until the late game. On the late game it will start to give away pieces, probably because it can't explore and simulate the correct branches due to the randomness involved.
- Finding different policies/evaluations that are effective (other than the piece ratio evaluation) is difficult, because the game hasn't many positional principles unlike Chess.
- The speed and effectiveness of the MCTS could be improved if it were implemented in other language that allows for more simulations per second.