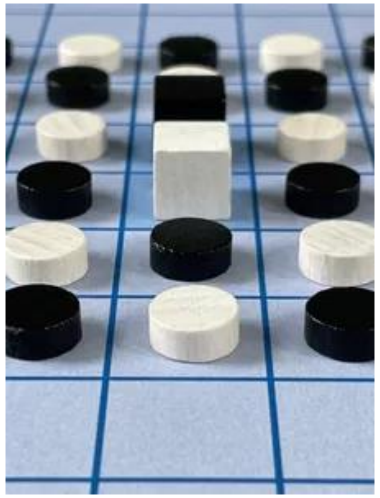
SPLINTER



Artificial Intelligence Project
Group 44_2D:

- Diogo Câmara - up201905166
- Mário Ferreira – up201907727
- Pedro Moreira - up201904642

Specification of the work to be performed

We choose Topic 2, which consists of Adversarial Search Methods for Two-Player Board Games, more specifically Splinter board game, which is an abstract strategy game for two players. The objective of the game is to get more connected board pieces to your king than your adversary is able to, this is to splinter your opponent’s king into a group that is smaller than your king’s group. The game is characterized by the type of board and pieces, the rules of movement of the pieces (operators) and the finishing conditions of the game with the respective score.

Game Related Work

The development of our game was done following the guidelines found in this link provided on this course Moodle Page, consisting of its components, lay-out and rules of the game.

Problem as a search problem

- State Representation and Initial State

  Play begins with the pieces arranged in a single, inter-connected group, with the kings occupying the board's two central squares. This initial state is to be represented by a 2d list in the Game Class.

- Objective test

  The game ends if a splinter causes the kings to occupy two different groups and the player whose king occupies the larger group wins the game. If both groups are of the same size, the game ends in a tie. If a move causes one of the kings to get pushed of the board, the other king's player wins.

- Operators

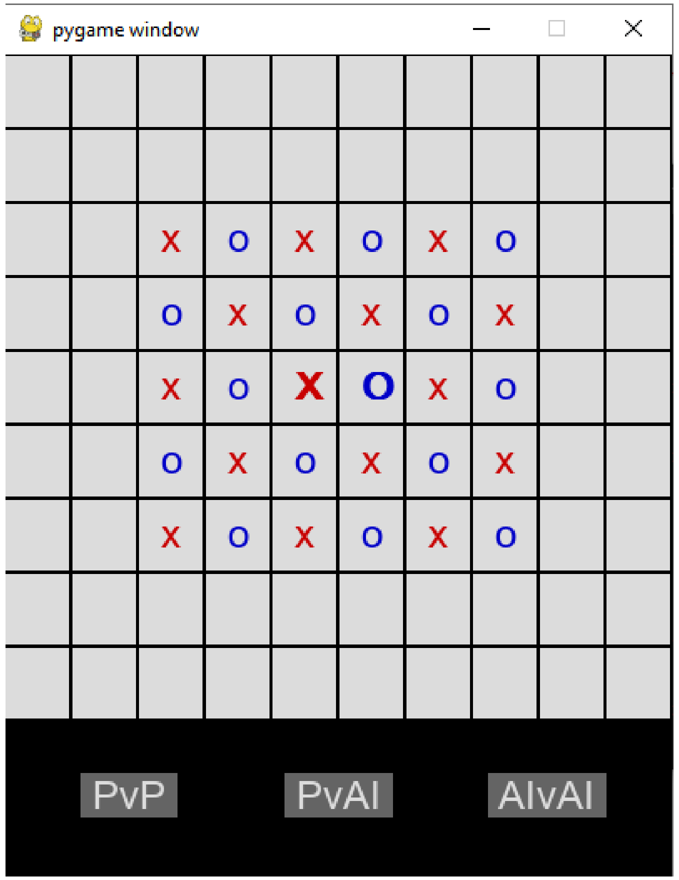
  The game has a unique operator with some restrictions such as:
  - you may move one of your own pieces one square in any direction (including diagonally) so that it lands on an adjacent square.
  - you may push a single piece or a row of connected pieces of either or both colors.
  - you may move or push any piece of either color off the board.
  - Kings may be moved (and may push and be pushed) in the same way as pawns.
  - One move can not fill the vacant spot, if any, left by the opponent's previous move

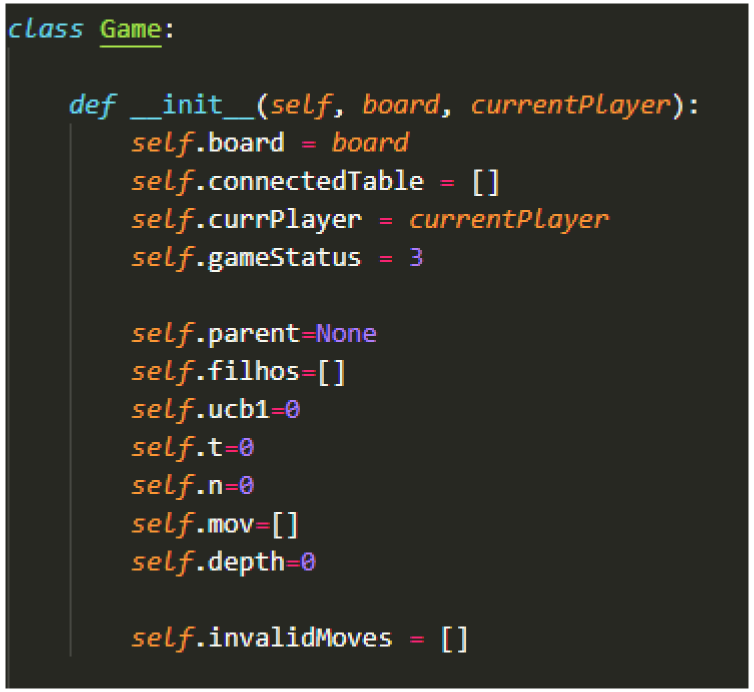
- Heuristics/Evaluation function

  If a splinter causes the kings to occupy two different groups, the game ends and the player whose king occupies the larger group wins the game.
  The heuristic chosen is based on counting the score of the pieces that are near the kings and the number of valid moves for each player, giving different weights to each of these attributes.

Implementation work

Our game is being developed on Python 3.10 . We used the pygame library to build the user interface for the game. We used a class to represent the game state, with various attributes such as the board, which consists of a 2d list containing the pieces, another attribute representing which player's turn it is, and so on.



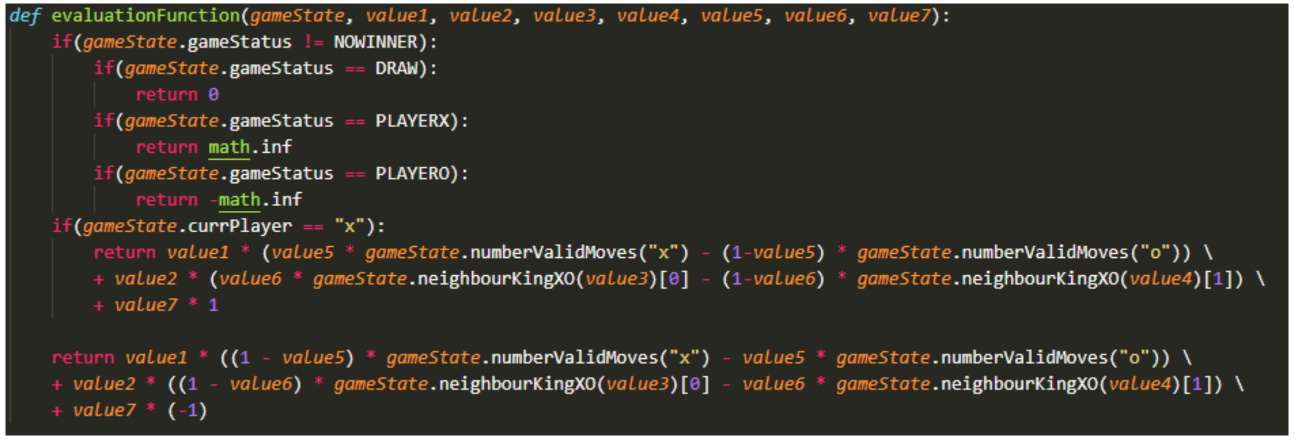


```
class Game:

    def __init__(self, board, currentPlayer):
        self.board = board
        self.connectedTable = []
        self.currPlayer = currentPlayer
        self.gameStatus = 3

        self.parent=None
        self.filhos=[]
        self.ucb1=0
        self.t=0
        self.n=0
        self.mov=[]
        self.depth=0

        self.invalidMoves = []
```

The Approach

Heuristic

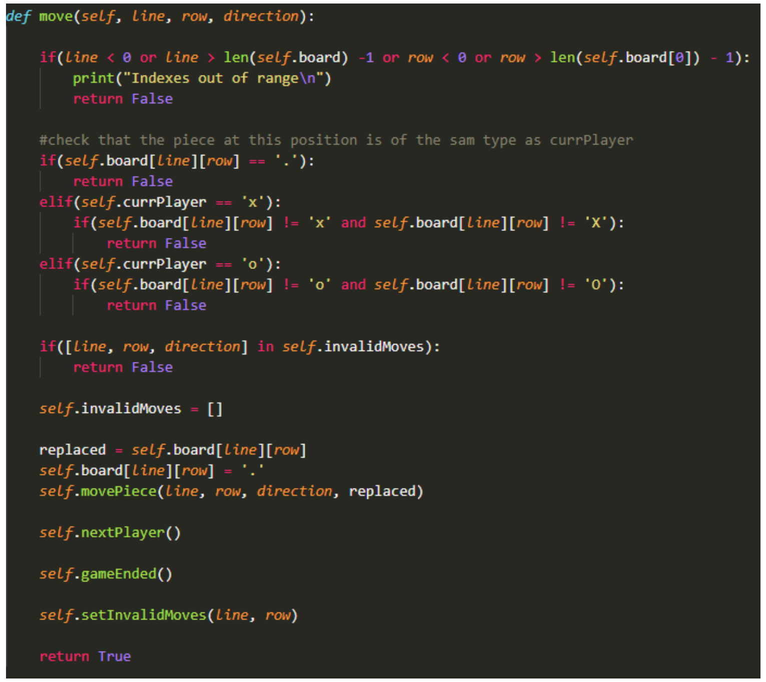
Our evaluation function maximizes values for the X player and minimizes for the O player. The function is based on the diference between the number of valid moves that each player has and the diference between the value of the pieces in each kings' neighbourhood. Each one of these values has a different weight, which is represented by the values in the image below.



```python
def evaluationFunction(gameState, value1, value2, value3, value4, value5, value6, value7):
    if(gameState.gameStatus != NOWINNER):
        if(gameState.gameStatus == DRAW):
            return 0
        if(gameState.gameStatus == PLAYERX):
            return math.inf
        if(gameState.gameStatus == PLAYERO):
            return -math.inf
    if(gameState.currPlayer == "x"):
        return value1 * (value5 * gameState.numberValidMoves("x") - (1-value5) * gameState.numberValidMoves("o")) \
        + value2 * (value6 * gameState.neighbourKingXO(value3)[0] - (1-value6) * gameState.neighbourKingXO(value4)[1]) \
        + value7 * 1

    return value1 * ((1 - value5) * gameState.numberValidMoves("x") - value5 * gameState.numberValidMoves("o")) \
    + value2 * ((1 - value6) * gameState.neighbourKingXO(value3)[0] - value6 * gameState.neighbourKingXO(value4)[1]) \
    + value7 * (-1)
```

Operator

We have a single operator "move", which receives the coordinates, line and column, and the direction to move the piece. Each instance of the game state has this operator. This way, it can check if the coordinates represente a valid piece to be played and if the move belongs to the invalid moves list, which is generated after each move.

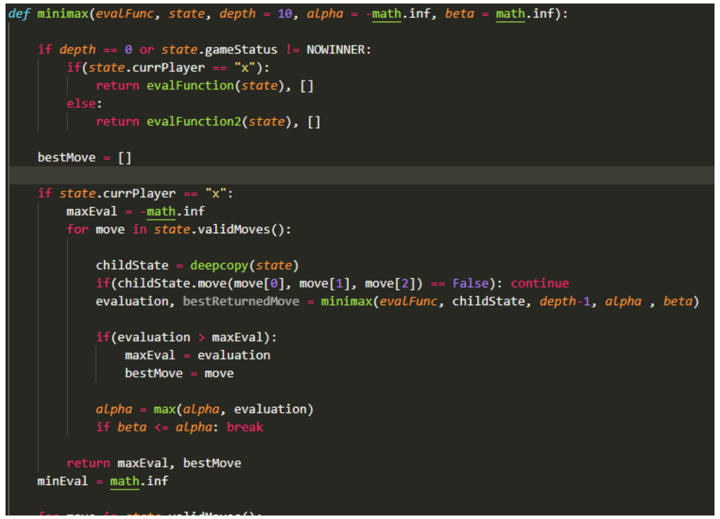


```python
def move(self, line, row, direction):

    if(line < 0 or line > len(self.board) -1 or row < 0 or row > len(self.board[0]) - 1):
        print("Indexes out of range\n")
        return False

    #check that the piece at this position is of the sam type as currPlayer
    if(self.board[line][row] == '.'):
        return False
    elif(self.currPlayer == 'x'):
        if(self.board[line][row] != 'x' and self.board[line][row] != 'X'):
            return False
    elif(self.currPlayer == 'o'):
        if(self.board[line][row] != 'o' and self.board[line][row] != 'O'):
            return False

    if([line, row, direction] in self.invalidMoves):
        return False

    self.invalidMoves = []

    replaced = self.board[line][row]
    self.board[line][row] = '.'
    self.movePiece(line, row, direction, replaced)

    self.nextPlayer()

    self.gameEnded()

    self.setInvalidMoves(line, row)

    return True
```

Implemented Algorithms

Minimax

Our version of the minimax algorithm tries to maximize the values for the X player and minimizing the values for the O player. Furthermore, to improve its efficiency, we are also using alpha-beta cuts to ignore some branches of the search tree.



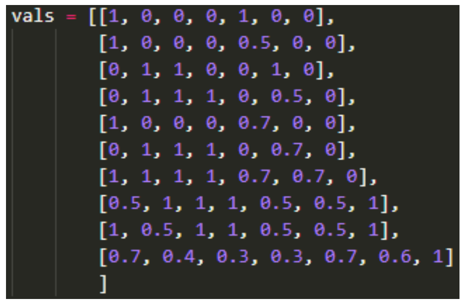
```python
def minimax(evalFunc, state, depth = 10, alpha = -math.inf, beta = math.inf):

    if depth == 0 or state.gameStatus != NOWINNER:
        if(state.currPlayer == "x"):
            return evalFunction(state), []
        else:
            return evalFunction2(state), []

    bestMove = []

    if state.currPlayer == "x":
        maxEval = -math.inf
        for move in state.validMoves():

            childState = deepcopy(state)
            if(childState.move(move[0], move[1], move[2]) == False): continue
            evaluation, bestReturnedMove = minimax(evalFunc, childState, depth-1, alpha , beta)

            if(evaluation > maxEval):
                maxEval = evaluation
                bestMove = move

            alpha = max(alpha, evaluation)
            if beta <= alpha: break

        return maxEval, bestMove
    minEval = math.inf

    for move in state.validMoves():
```

Monte Carlo

Our version of Monte Carlo algorithm works in 4 phases. Phase one is a tree traversal using ucb1 formula. Second phase is a tree expansion where you add new nodes into the tree. Phase tree is a rollout where you do a random simulation of the game. Finally, there is backpropagation where you use the values from the rollout and put them at our current possible nodes. This values will be used to chose the best move in our current problem situation.

Genetic Algorithm

We decided to implement a genetic algorithm to improve the evaluation function used in the Minimax and Monte Carlo algorithms. It start with a given initial population of 10 lists of values to be used in the evaluation function and performs games using different heuristic values. In the end, it selects the 5 most victorious lists of heuristic values and uses them to create the next population, repeating this process 5 times.



```python
vals = [[1, 0, 0, 0, 1, 0, 0],
        [1, 0, 0, 0, 0.5, 0, 0],
        [0, 1, 1, 0, 0, 1, 0],
        [0, 1, 1, 1, 0, 0.5, 0],
        [1, 0, 0, 0, 0.7, 0, 0],
        [0, 1, 1, 1, 0, 0.7, 0],
        [1, 1, 1, 1, 0.7, 0.7, 0],
        [0.5, 1, 1, 1, 0.5, 0.5, 1],
        [1, 0.5, 1, 1, 0.5, 0.5, 1],
        [0.7, 0.4, 0.3, 0.3, 0.7, 0.6, 1]
        ]
```

Conclusions

This project gave us an insight into some algorithms that can be used to simulate a player in a turn based game. It presented us with some challenges, such as the formulation of a good evaluation function and optimization of the time efficiency of the diffenrent algorithms.

To conclude, we believe that the project goals were fulfilled and it helped to consolidate the contentes presented in the lectures.

References

- Splinter Rules https://splinterboardgame.blogspot.com/2021/06/splinter-is-two-player-abstractstrategy.html
- Monte Carlo Tree Search https://www.youtube.com/watch?v=UXW2yZndl7U