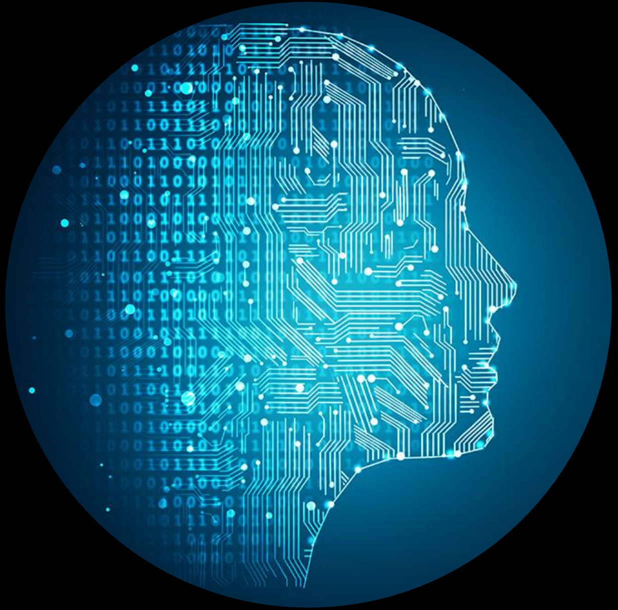
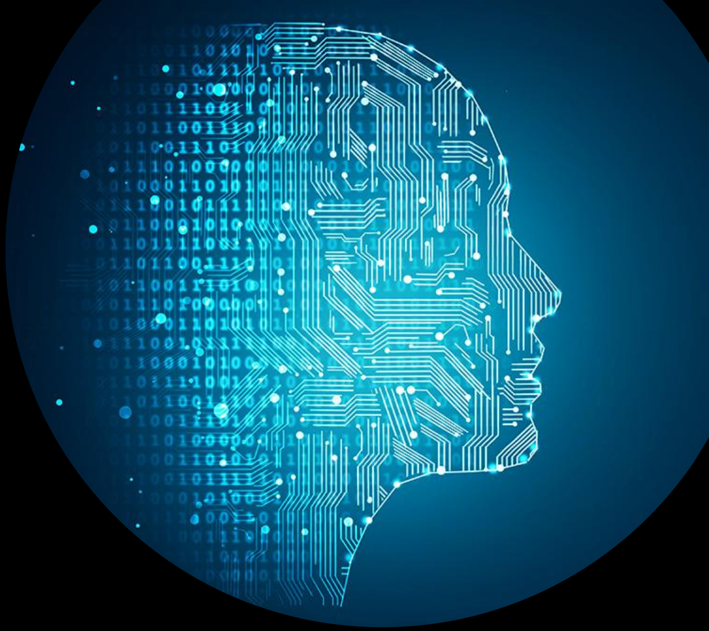
AI project 1
CIFRA CODE25

Checkpoint 1

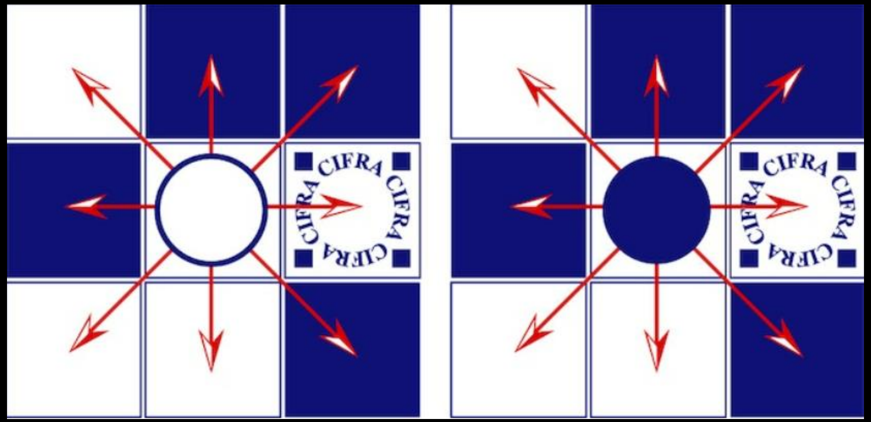
(1) - Definition of the game

The game to be implemented is the CIFRA Code25, a 2-player strategy board game played in a 5x5 board with 25 randomly placed white and blue tiles that form the board.

The first player to play is arbitrarily decided by the players, and is who plays first, and the second player has the right to choose one of the 4 sides of the board to play and the color of his/her pieces. The opposite side is automatically first player's side and he plays with the pieces of the other color.

Each player has 5 pieces of his/her respective color. The main objective of the game is to move one's pieces to the opposite side of the board, but pieces can also be captured (when a piece arrives the opposite side, it is called a goal piece, and it can't be moved or captured anymore). A piece is captured when a player moves one of his/her pieces to the same tile of an opponents piece.

The game ends when all available pieces of a player are goal or when all his/her not goal pieces are captured. The player who manages to get all his available pieces to align in the opposite side of the board faster is considered the winner. If there is a tie, the player who has more available (not goal) pieces alive on the board wins.



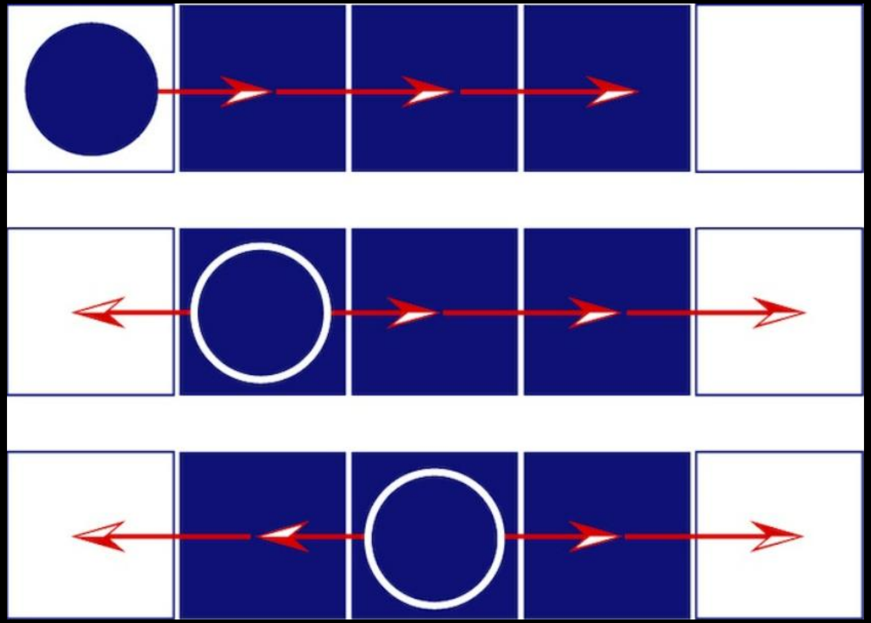


Fig 1,2- Ilustration of movimentation

(2) - Bibliography

https://www.kickstarter.com/projects/logygames/cifra-code25

(3) - Formulation as a search problem

- **State representation:**

Matrix (2D array) with board, board[5,5], or, in general, board[N,N], filled with three different values, either: (255, 255, 255)- RGB code for white- if a white piece is occupying the tile; (0, 0, 255))- RGB code for blue- if a blue piece is occupying the tile; 0, if the tile is not being occupied. Next player to play the game is represented by a variable turn.

- **Initial state**

There are (24C12)/4=676,039 initial states. One of them is represented on the image on the right.

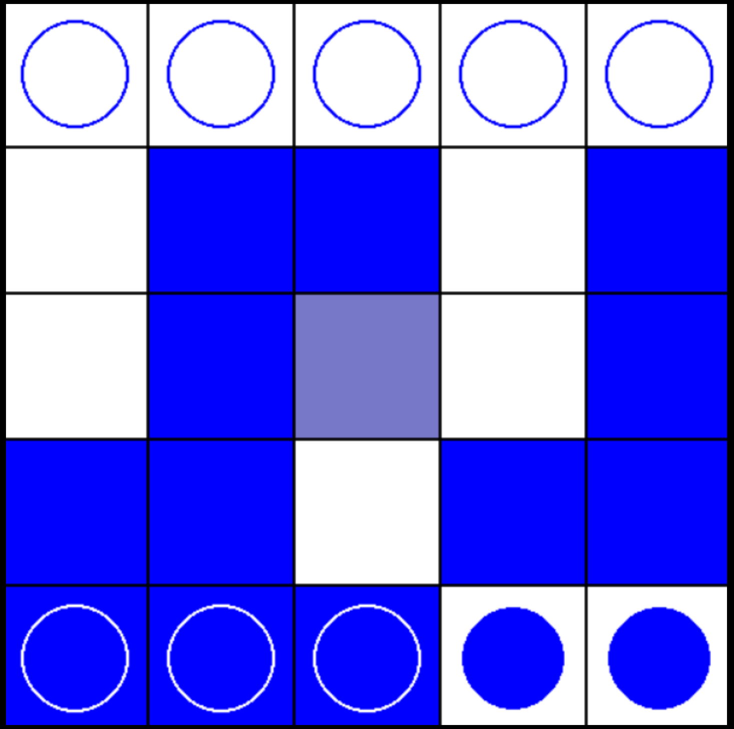


Fig 3- Example of an initial game state

(3) - Formulation as a search problem

- **Objective test:**

```
//WHITE- win for player with the white pieces
//BLUE- win for the player with the blue pieces
//None- game not yet finished
```

```
def winner(self):
    if self.white_goal > self.blue_goal and self.white_left == self.white_goal:
        return WHITE
    elif self.blue_left <= 0:
        return WHITE

    if self.blue_goal > self.white_goal and self.blue_left == self.blue_goal:
        return BLUE
    elif self.white_left <= 0:
        return BLUE

    return None
```

- **Operators:**

  - **Name:**

    There are 8 possible moves for each piece: up, down, left, right, up-left, up-right, down-left, down-right, and the number of tiles that a piece can move according to this directions is specified by the movement rules already shown in Fig 2.

```
possible_moves.update({(piece.row, left), (piece.row, right), (up, piece.col), (down, piece.col), (up, left), (up, right), (down, left), (down, right)})
```

(3) - Formulation as a search problem

Operators:

- **Pre-conditions:**

  For a piece to move to a certain tile, it has to be empty or occupied by a (not goal) adversarial piece (so it can capture it). Obviously, each of the coordinates (row, col) of the tiles have to be between 0 and the number of rows minus 1 (number of rows is always equal to the number of columns, as the board is always a square)

```
if((0 <= row < ROWS) and (0 <= col < ROWS)):
    if self.board.get_piece(row, col) == 0 or (self.board.get_piece(row, col).color != self.turn and
self.board.get_piece(row, col).goal == False):
        valid_moves.add(move)
```

- **Effects:**

```
def move(self, piece, row, col):
    self.pieces[piece.row][piece.col], self.pieces[row][col] = self.pieces[row][col], self.pieces[piece.row][piece.col]
    piece.move(row, col)
```

- **Cost:**

  Each step costs 1, being the cost of the solution the number of steps to solve the problem

(4) - Implementation work

- Programming language: Python with PyGame library to implemente a GUI
- The game will have three game modes: Human vs Human; Human vs Computer and Computer vs Computer
- Human player will be able to choose Computer difficulty: easy, medium, or high (according to minimax depth)
- Data structures: A 2D array is being used to represent the game board (in a matrix format)
- Minimax with alpha-beta cuts and its variants is going to be used, alongside Monte Carlo search trees, to make the computer decide the best actions at each playing point according to the defined heuristics

- The Human vs Human game mode is already implemented, i.e., the game is already fully playable between two human players. We're now going to focus our efforts in the AI implementation to get the other two above referenced game modes done

Implementation work already carried out:

The game is already implemented, i.e., the Human vs Human game mode
Thus, from now on we're going to focus on the implementat

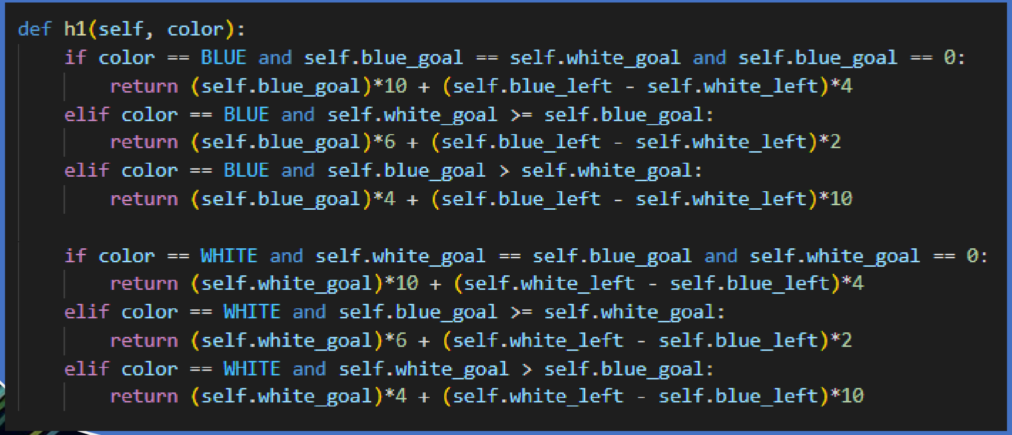
(5) - Approach (heuristics/evaluation functions)

- Heuristics refer to problem-solving strategies or techniques that people use to make quick, efficient, and intuitive judgments or decisions. While heuristics can be helpful in making decisions quickly, they can also lead to errors and biases in judgment. In our game we used two heuristics:

- The first heuristic (h1) attempts to estimate the potential value of a given state in the game by considering both the progress that has been made towards the primary objective of the game (goal pieces count) and the overall strength of each player's position on the board (piece count).

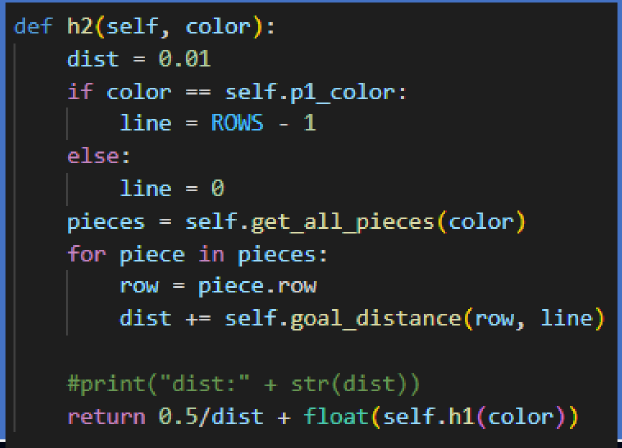
- The second heuristic (h2) attempts to estimate the potential value of a given state by taking into account the first heuristic and the third.

- The second heuristic (h3) attempts to estimate the potential value of a given state by taking into account the distance of the player's pieces from the player's goal line.



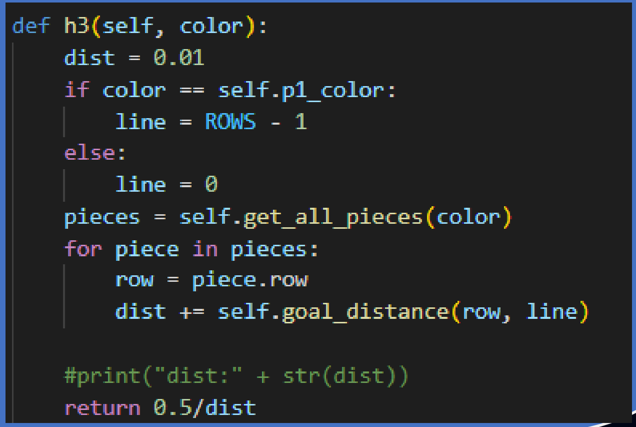
```python
def h1(self, color):
    if color == BLUE and self.blue_goal == self.white_goal and self.blue_goal == 0:
        return (self.blue_goal)*10 + (self.blue_left - self.white_left)*4
    elif color == BLUE and self.white_goal >= self.blue_goal:
        return (self.blue_goal)*6 + (self.blue_left - self.white_left)*2
    elif color == BLUE and self.blue_goal > self.white_goal:
        return (self.blue_goal)*4 + (self.blue_left - self.white_left)*10

    if color == WHITE and self.white_goal == self.blue_goal and self.white_goal == 0:
        return (self.white_goal)*10 + (self.white_left - self.blue_left)*4
    elif color == WHITE and self.blue_goal >= self.white_goal:
        return (self.white_goal)*6 + (self.white_left - self.blue_left)*2
    elif color == WHITE and self.white_goal > self.blue_goal:
        return (self.white_goal)*4 + (self.white_left - self.blue_left)*10
```



```python
def h2(self, color):
    dist = 0.01
    if color == self.p1_color:
        line = ROWS - 1
    else:
        line = 0
    pieces = self.get_all_pieces(color)
    for piece in pieces:
        row = piece.row
        dist += self.goal_distance(row, line)

    #print("dist:" + str(dist))
    return 0.5/dist + float(self.h1(color))
```

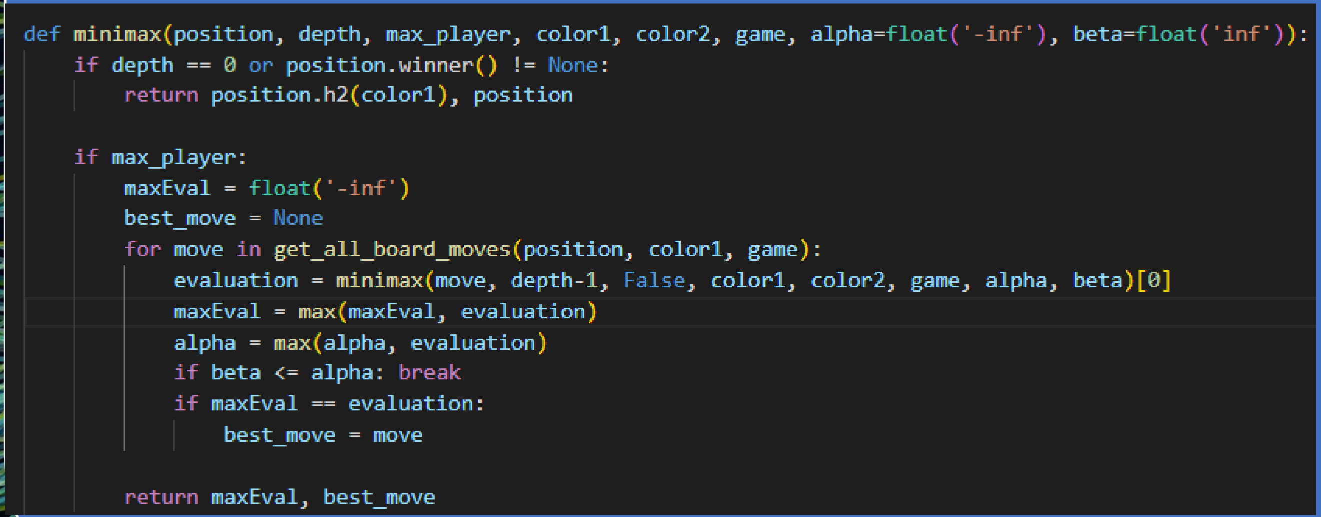


```python
def h3(self, color):
    dist = 0.01
    if color == self.p1_color:
        line = ROWS - 1
    else:
        line = 0
    pieces = self.get_all_pieces(color)
    for piece in pieces:
        row = piece.row
        dist += self.goal_distance(row, line)

    #print("dist:" + str(dist))
    return 0.5/dist
```

(5) - Approach (heuristics/evaluation functions)

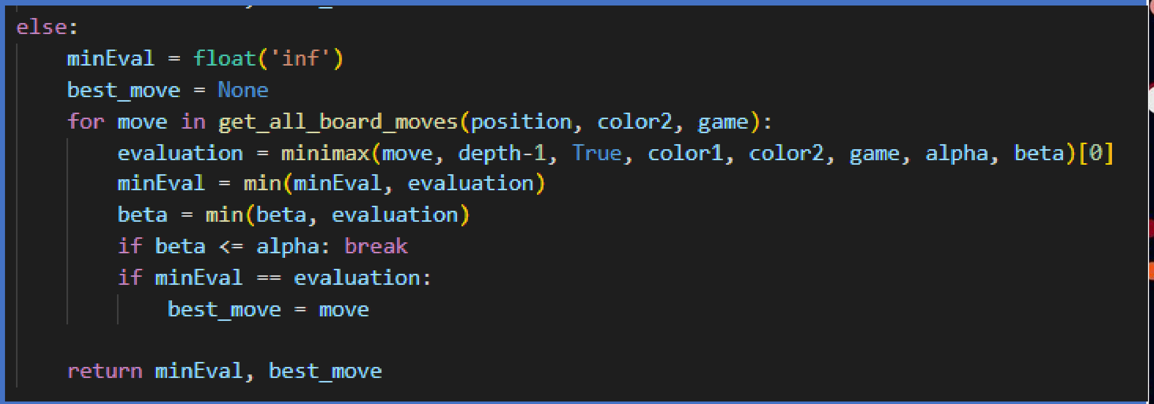
- In the context of algorithms, an evaluation function is a mathematical function that is used to assess the quality of a particular state or solution. The evaluation function is often used in heuristic search algorithms, such as Minimax in our case, to evaluate the potential value of a particular state or solution.



```python
def minimax(position, depth, max_player, color1, color2, game, alpha=float('-inf'), beta=float('inf')):
    if depth == 0 or position.winner() != None:
        return position.h2(color1), position

    if max_player:
        maxEval = float('-inf')
        best_move = None
        for move in get_all_board_moves(position, color1, game):
            evaluation = minimax(move, depth-1, False, color1, color2, game, alpha, beta)[0]
            maxEval = max(maxEval, evaluation)
            alpha = max(alpha, evaluation)
            if beta <= alpha: break
            if maxEval == evaluation:
                best_move = move

        return maxEval, best_move
```



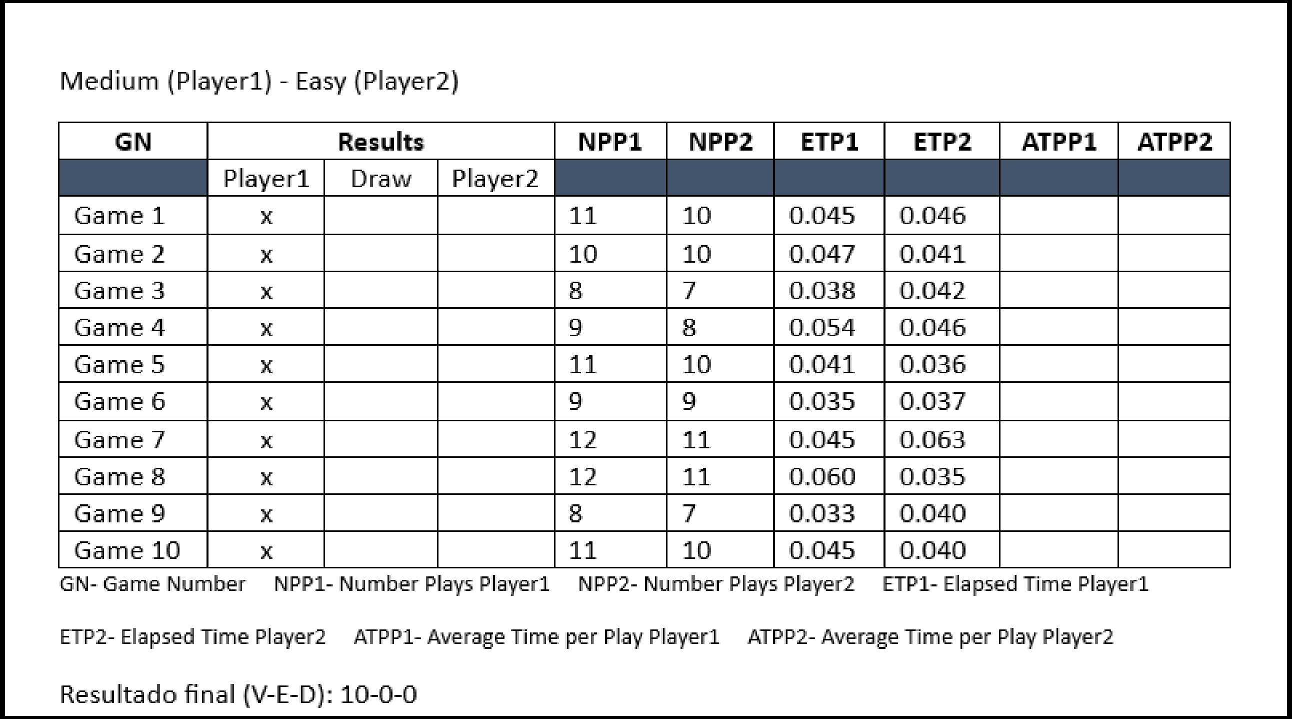
```python
else:
    minEval = float('inf')
    best_move = None
    for move in get_all_board_moves(position, color2, game):
        evaluation = minimax(move, depth-1, True, color1, color2, game, alpha, beta)[0]
        minEval = min(minEval, evaluation)
        beta = min(beta, evaluation)
        if beta <= alpha: break
        if minEval == evaluation:
            best_move = move

    return minEval, best_move
```

(6) - Implemented algorithms (search algorithms, minimax, metaheuristics)

- In this project we used Minimax.
- Minimax is a decision-making algorithm used in game theory and artificial intelligence to determine the best move that a player should make. The algorithm involves evaluating all possible moves available to a player and assigning a score to each move based on the potential outcomes of the game. The player then selects the move with the highest score. The name "minimax" comes from the two components of the algorithm: minimizing the potential loss if the opponent makes the best possible move, while maximizing the potential gain for the player. We implemented the minimax as we saw before using the defined heuristics.

(7) - Experimental results



Medium (Player1) - Easy (Player2)

GN	Results			NPP1	NPP2	ETP1	ETP2	ATPP1	ATPP2
	Player1	Draw	Player2						
Game 1	x			11	10	0.045	0.046		
Game 2	x			10	10	0.047	0.041		
Game 3	x			8	7	0.038	0.042		
Game 4	x			9	8	0.054	0.046		
Game 5	x			11	10	0.041	0.036		
Game 6	x			9	9	0.035	0.037		
Game 7	x			12	11	0.045	0.063		
Game 8	x			12	11	0.060	0.035		
Game 9	x			8	7	0.033	0.040		
Game 10	x			11	10	0.045	0.040		

GN- Game Number NPP1- Number Plays Player1 NPP2- Number Plays Player2 ETP1- Elapsed Time Player1

ETP2- Elapsed Time Player2 ATPP1- Average Time per Play Player1 ATPP2- Average Time per Play Player2

Resultado final (V-E-D): 10-0-0

The other tables are in a separate file

Conclusions

Player 1 vs Player 2 anaylisis:

- Medium(h3) vs Easy(random): similar elapsed time for both and similar number of plays, but medium mode won always vs easy mode
- Hard(h2) vs Easy(random): elapsed time superior for hard and similar number of plays for both, but hard mode won always vs easy mode
- Hard(h2) vs Medium(h3): elapsed time superior for hard and similar number of plays for both, but hard mode won most of the times vs medium mode
- Easy(random) vs Medium(h3): similar elapsed time for both and similar number of plays, but medium mode won always vs easy mode
- Easy(random) vs Hard(h2): elapsed time superior for hard and similar number of plays for both, but hard mode won always vs easy mode
- Medium(h3) vs Hard(h2): elapsed time superior for hard and similar number of plays for both, but hard mode won most of the times vs medium mode

The performance and time required for different heuristics when using the minimax algorithm to search a game tree can vary significantly depending on the specific game being played and the quality of the heuristic. Looking at the results we can conclude that the best heuristic matches the hard mode, and then goes the medium mode, and for last the easy mode which is random. Although the hard mode has the best heuristic in terms of evaluating the state of the board, it requires more computacional power than the others.

References

https://moodle.up.pt/course/view.php?id=2176

https://chat.openai.com/

https://www.geeksforgeeks.org/

https://ai-boson.github.io/mcts/

https://stackoverflow.com/

https://labs.openai.com/

END