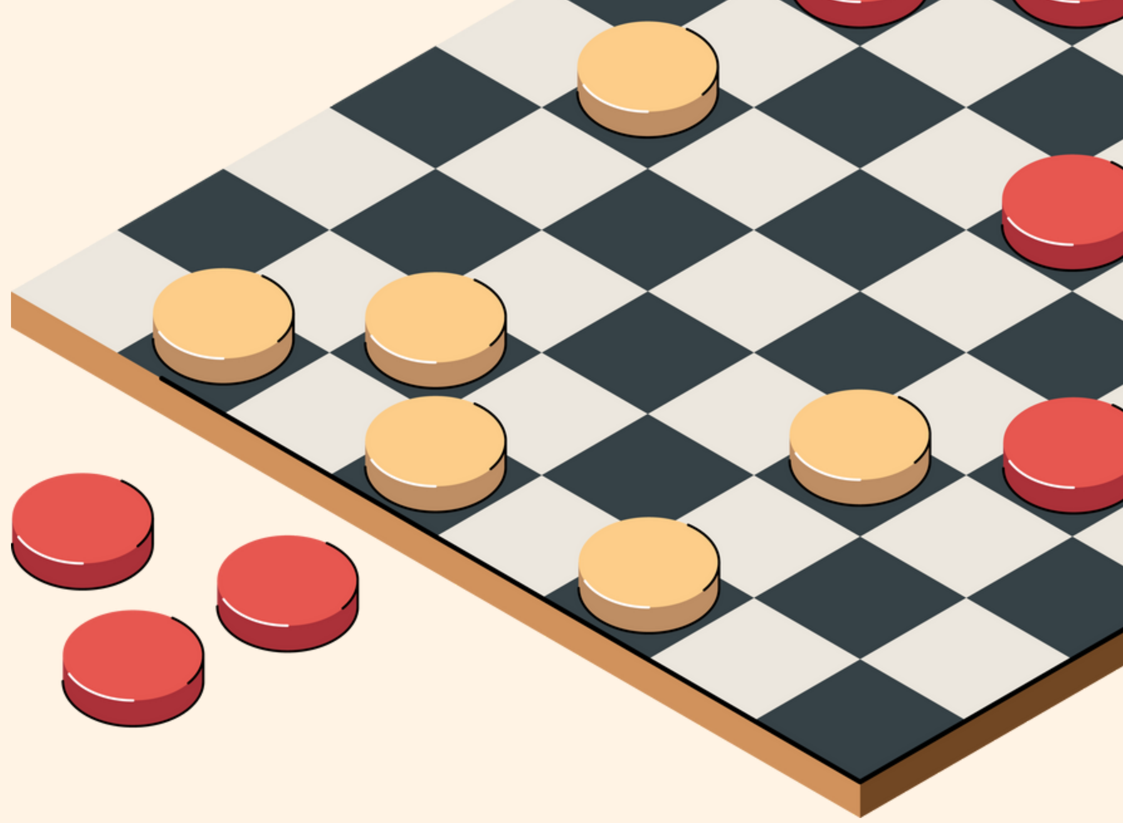
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

FOCUS

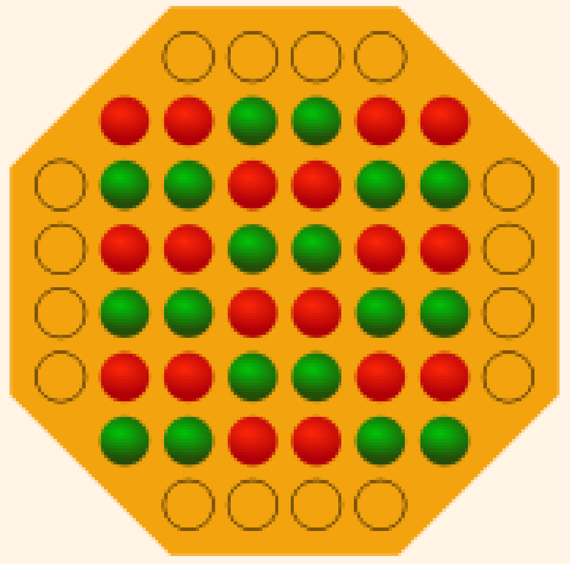
Francisca Guimarães - 202004229
Inês Oliveira - 202103343
Miguel Santos - 202008450

Final Presentation



WORK SPECIFICATION

The objective of our project is to design and implement adversarial search methods for a strategic game titled "Focus". The game is played by two to four players on a 6x6 board with 1×4 extensions on each side (8x8 board). This alteration results from removing the three squares in each of the board's corners, creating a distinct playing field that challenges conventional checkerboard strategies.

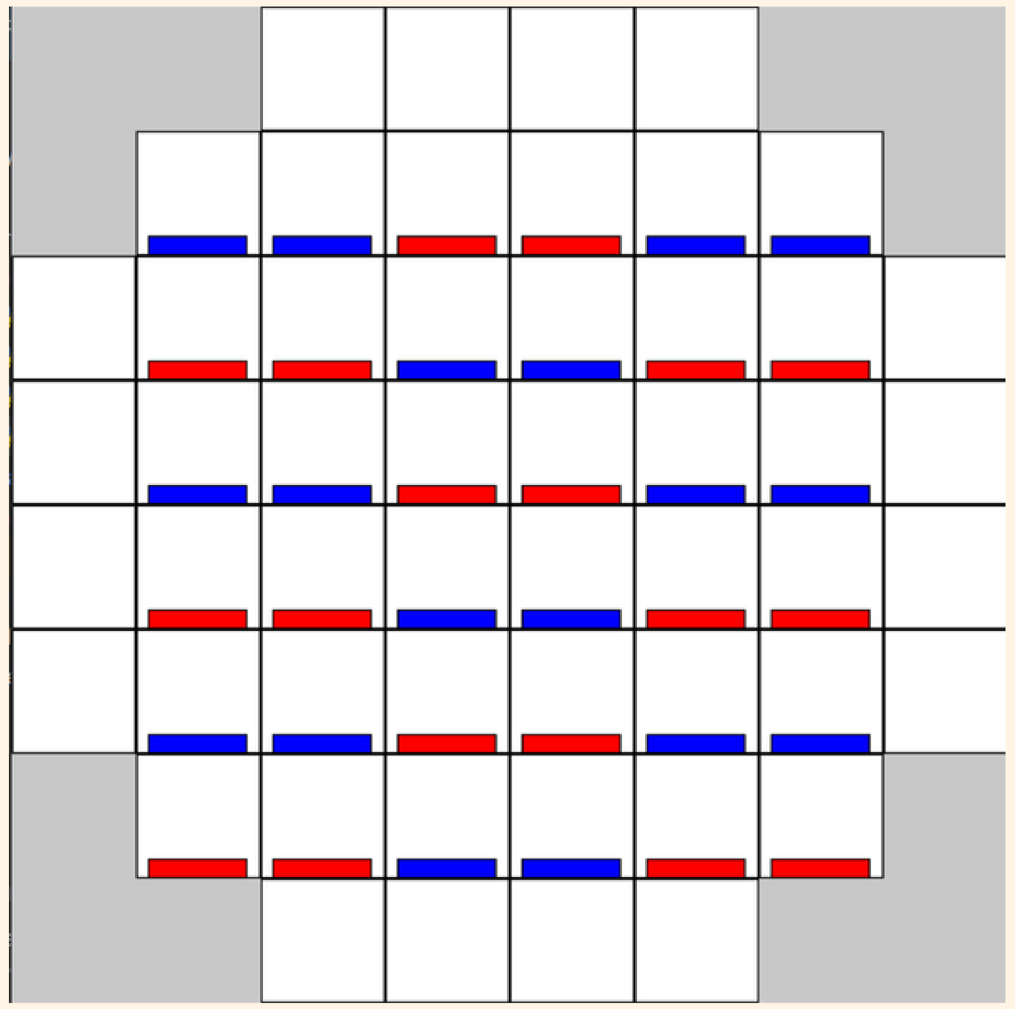


In the game Focus, players navigate stacks of one to five pieces on a board, with each stack able to move spaces equal to its height. A player can only move a stack if they own the top piece. The game's layering system allows stacks to merge when they land on each other, ensuring strategic depth. If a merged stack exceeds five pieces, the excess is removed from the bottom until it meets the game's limit, with removed pieces either captured or reserved for later use, based on the piece's ownership. This mechanic emphasizes strategy in piece movement and stack management.

RELATED WORK

To start the development of the game, we did not consult any similar game, always using the documentation available at the link: https://en.wikipedia.org/wiki/Focus_(board_game), which contains a pdf with all the relevant information for development (https://www.hasbro.com/common/instruct/domination.pdf).

Additionally, to understand the dynamics of the game, we used the content available at: https://mindsports.nl/index.php/the-pit/529-focus to understand the movement of the pieces, since it is possible to play the game or watch computer vs computer.



FORMULATION OF THE PROBLEM

Initial State: Each player pieces are in the original positions as documented by the game (previous slide's picture). Player's turns vary.

State representation: The game state can be represented as a matrix of 8x8. "R" represents the player with the red pieces. "B" represents the players with the blue pieces. "X" the places where there are currently no pieces. "N" the non-existant places pieces because they don't belong to the actual board. Also, the number of reserved pieces is saved during the course of the game.

Objective test: Check if player cannot move a stack or piece on the gameboard or has no reserve pieces. The winner is the player whose pieces are the only ones on the board.

Operators:

Name	Pre-conditions	Effects	Cost
Stack move	• A move can only be made in a straight line, either vertically or horizontally (never diagonally). • You can only move a playing piece of your own color. • You can only move a stack that has your color playing piece on top. • The number of pieces in the stack you wish to move determines the max number of spaces you may move that stack.	• The position of the piece is changed, forming or adding to a stack • The position of a stack is changed, potentially removing player's pieces from the bottom of the stack (max 5 pieces) and capturing pieces to form a reserve	1
Reserve piece move	• You must have a reserve piece available to play.	Adds a piece to the board, changing the game's dynamic in your favour	1

HEURISTICS/EVALUATION FUNCTIONS

Our search algorithm will rely on evaluation functions to assess the quality of the current game state.
To formulate the evaluation function, we developed several heuristic methods:

Piece Count Heuristic: Counts the number of pieces a player has on the board compared to their opponent. The more pieces a player has, the better their position.

Piece Mobility Heuristic: Measures a player's mobility, i.e., how many options a player has for moving pieces. More options generally mean more control and the ability to respond to the opponent's moves.

Stack Control Heuristic: Evaluates the control of stacks. Since controlling stacks is a crucial part of the game, stacks that are controlled by a player, especially if they contain pieces, that could potentially be moved to dominate other stacks, are more valuable.

Board Position Heuristic: Pieces in positions to make a capture or to support other pieces are valued higher.

Reserve Management Heuristic: Reserve pieces are valuable as they allow for more flexibility and can significantly alter the state of play. Takes into account the number of reserve pieces a player has, as well as the potential these reserves have to affect the board (like turning the tide of a weak position or breaking up a large stack).

Each heuristic is allocated distinct weights to prioritize certain aspects over others.

FINAL WORK IMPLEMENTED

To develop the project we chose the Python language, using the PyGame library and the PyCharm IDE.

At the game's start, you're greeted with a main menu offering options to either start playing or view instructions. Choosing to start launches into a selection for board size (6x6 or 8x8).

Following that, you pick a play mode: playing against another player, against the computer, or watching the computer play against itself. If you're playing (either against another player or the computer), you'll choose which color you start the game with.

In matches involving the computer, you also set the difficulty level: easy, medium, hard, or expert, affecting the computer's strategy and challenge level. These levels represent different AI strategies, ranging from simple Monte Carlo Search for easy mode to more complex Minimax algorithms with varying depths and evaluation functions for harder modes.

Additionally, the game always displays all possible moves to help guide your decisions. And if you're playing and take a while to move, the game offers helpful hints to keep things moving.

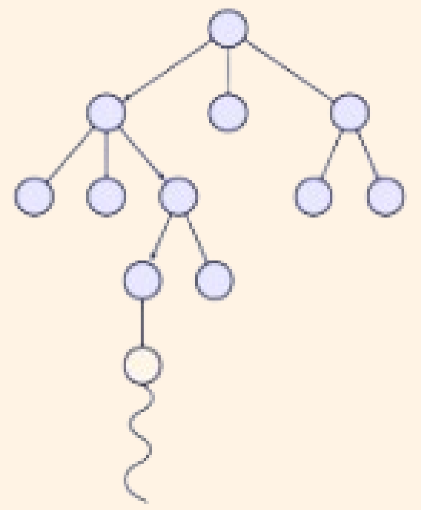
MONTE CARLO TREE SEARCH

Builds a search tree over time, with nodes representing game states and edges representing moves. Starting from the current game state, it repeatedly simulates games, making random moves until reaching a terminal state (win, lose, or draw). Each simulation contributes to a statistical model of the decision space, which MCTS uses to make increasingly informed decisions about which moves are likely to lead to a win.

The process consists of four phases:

- Selection: Starting from the root node (the current game state), it selects child nodes down the tree based on a balance between exploration (trying unexplored moves) and exploitation (choosing moves with a high win rate), using the Upper Confidence Bound (UCB1) formula for trees, also known as the UCT score.
- Expansion: Upon reaching a node that is not fully expanded (not all possible moves have been tried), it expands the tree by creating a new child node for one of the unexplored moves.
- Simulation: From the new node, it simulates a random play-out to a terminal state.
- Backpropagation: It updates the nodes from the played-out simulation back up the tree, increasing the visit count for each node and updating the win count based on the simulation's outcome.

The Monte Carlo algorithm struggles to find the best moves because it doesn't have any light-weight heuristics to guide its decisions. Our game has many possible moves, making it even harder for the algorithm to pick good ones with limited tries. As a result, it often seems to play randomly throughout the game, only occasionally, and in test case scenarios, making smart moves when there are only a few pieces left. That's why we consider it to be at an easy level of difficulty.



MINIMAX WITH ALPHA BETA CUTS

The goal of the Minimax algorithm is to maximize the current player's advantage (maximizingPlayer) and minimize the opponent's, alternating between these two modes at each level of the tree. When the specified depth is reached or the game comes to an end (*game_logic.check_gameover()*), the algorithm evaluates the state of the game using the *evaluate()* function. For a maximizing player (*maximizingPlayer*), it searches for the move that leads to the highest evaluation value, considering all valid moves. For the opponent, it looks for the move that results in the lowest value, simulating a defensive strategy. This is done recursively, decreasing the depth with each call, until the base of the decision tree is reached, allowing the algorithm to decide on the best possible move from the current state of the game.

After this implementation was complete, we decided to add the alpha beta cuts. By adding this, we can prune unproductive branches from the search tree, which improves efficiency without sacrificing the quality of the decision. This allows for faster response times even on an 8x8 board with many pieces, as it avoids evaluating states that will not influence the final result.

On the other hand, when the algorithm plays against itself (same depth and evaluation function) it is common for it to enter a cycle and for the game not to progress. To avoid this problem, we ensure that when there are several moves with the same score (best score), the move is chosen randomly, guaranteeing an optimal but non-deterministic solution.

APPROACH

To improve our evaluation process, we've merged the previously mentioned evaluation metrics with weights. These functions now consider:

H1 The *evaluate_medium* function calculates scores based on strategic positioning, particularly focusing on the control of central areas on the board. It assigns points for player pieces located in key central squares, identified dynamically based on board size (6x6 or 8x8). It emphasizes the importance of occupying central positions (**Board Position Heuristic**) for optimal gameplay.

H2 The *evaluate_hard* function evaluates the state of the game by assigning scores based on the total of the board stacks, reserved pieces (**Reserve Management Heuristic**) and captured pieces (**Piece Count Heuristic**). For each stack controlled (**Stack Control Heuristic**), points are added or subtracted depending on whether the controller is the player in question or their opponent. Pieces set aside and captured by the player also contribute positively to their total score.

EXPERIMENTAL RESULTS

Once we had implemented the two algorithms, we tested their efficiency against each other to find the best performers. For that, we measured the number of game plays, the average search time and the games result.

BOARD 6X6 *			
Minimax	H1D2 (MEDIUM)	H2D2 (HARD)	H2D5 (EXPERT)
H1D2 (MEDIUM)	-	20 | 0,0048 | 0-5	17 | 2,35 | 0-5
H2D2 (HARD)	-	-	48 | 2,17 | 0-5
H2D5 (EXPERT)	-	-	-

* NR. OF GAME PLAYS | AVG. SEARCH TIME (s) | RESULT (row - col)

BOARD 8x8 *			
Minimax	H1D2 (MEDIUM)	H2D2 (HARD)	H2D5 (EXPERT)
H1D2 (MEDIUM)	-	42 | 0,0324 | 0-5	38 | 53,4 | 0-5
H2D2 (HARD)	-	-	47 | 68,7 | 2-3
H2D5 (EXPERT)	-	-	-

* NR. OF GAME PLAYS | AVG. SEARCH TIME (s) | RESULT (row - col)

We started by testing minimax in a 6x6 board, with the two heuristics and with depth 2 and 5. We concluded that the second heuristic gives better results than the first and that the greater the depth, the greater the efficiency.

We did the same analysis for the 8x8 board, where we can see that the times are much higher and that in expert vs hard mode the game lasts much longer than the others.

The Monte Carlo Tree Search algorithm was also tested taking into consideration varations of the number of iterations the algorithm requires.

MCTS (Level Easy) vs Minimax H2D2 (Level Hard)				
	Nr. Iterations	Avg. Search Time (s)	Nr. Of Game Plays	Result
6x6	10	1.96	6	0-5
	30	6.5	7	0-5
	50	10.3	8	0-5
8x8	10	27,8	12	0-5
	30	76,4	18	0-5
	50	174	19	0-5

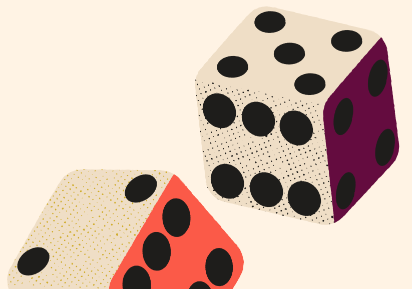
CONCLUSIONS

The development of the focus game has been successfully completed, as all the mandatory features and a few extra ones have been implemented. With the development of the game, we learned how to apply AI algorithms and how they influence the game and the player's next moves.

- By applying Minimax with alpha beta cuts with two different depths and two different evaluation functions, we can see that the more strong heuristics the algorithm has, the better results it gets.
- Applying different depths may slow down the algorithm, but it makes it better and more efficient, which means the game finishes faster.
- Monte Carlo Tree Search algorithm relies on random simulations. Based on the outcomes (total losses), it's definitely not ideal for our game's complexity. More pieces (8x8 board) mean longer response times, as it simulates all states. With more iterations, MCTS needs more moves to lose, exploring scenarios better but taking more time. Fewer pieces mean quicker responses; more iterations mean better move selection, but for this scenario, for that many iterations required to win, the game would be unplayable.

REFERENCES

https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/
https://towardsdatascience.com/monte-carlo-tree-search-an-introduction-503d8c04e168
https://www.hasbro.com/common/instruct/domination.pdf
https://mindsports.nl/index.php/the-pit/529-focus



See you at the next game!

THANKS FOR PLAYING