Artificial Intelligence
Assignment 1 – Final Delivery

Group 07

Carolina Rosemback Guilhermino, up201800171

José Eduardo Henriques, up201806372

Miguel Carreira Neves, up201608657

Project Specification

Two-Player Adversarial Board Game: Shobu

Turn based game, where each turn is comprised of two moves: first one Passive move and then one Aggressive move.

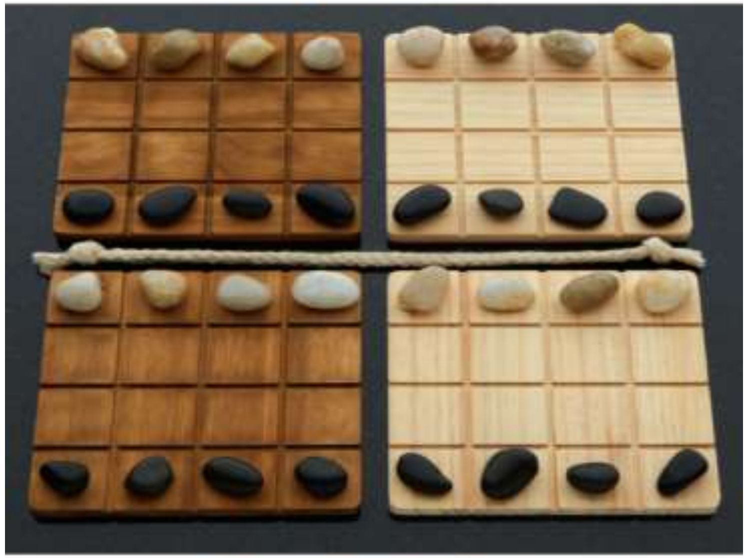
The passive move must be played on one of the player's two homeboards. The player chooses one of their colour pieces and moves it into any direction inside the board, up two spaces, without pushing or jumping over any piece.

The aggressive move must be made in the same direction and number of spaces as the passive move, on one of the opposite colour boards as the one chosen in the passive move. Additionally, the aggressive move can push, at most, one piece, of the opponent colour. If a piece is pushed off the board, that piece is removed from the game.

The game's objective is to remove all opponent pieces from one board. First one to do so wins the game.

In this project, the aim is to implement this game with PvP, PvC and CvC modes. The Computer should be provided with an AI, using Minimax search methods with different depth and $\alpha\beta$ cuts, ensuring different difficulty levels.

WHITE's Homeboards



BLACK's Homeboards

figure 1- game board

Search Problem Formulation

State Representation:

4-Dimensional matrix M[H[B[4,4], B[4,4]], H[B[4,4], B[4,4]]]. State M is a matrix consisting of two H matrices. H represents a player's homeboard, consisting of two B matrices. B represents a board, consisting of a 4x4 matrix. A board is filled with 'B', 'W' or ' ' chars, representing a black piece, a white piece and an empty space, respectively.

Initial State:

Each board's top row is filled with white pieces, bottom row is filled with black pieces and the rest with empty spaces (as shown in Fig. 1)

Objective State:

Any state containing a board with only black pieces (and empty spaces), assuming the black player's perspective.

Operators:

updateBoard(passive_piece, aggressive_piece, offset, piece, other_piece)

Operator Preconditions:

Both functions legalPassiveMoves and legalAgressiveMoves must return non-empty amount of options so that a turn can be considered valid.

Search Problem Formulation

Operator Preconditions (continuation):

“legalPassiveMoves”, returns which pieces can perform a passive move, for a given movement and board colour.

```
def legalPassiveMoves(self, color_side, row_index, col_index):
    options = []
    for i in range(row_index - 2, row_index + 3):  # 2 rows behind, 2 rows ahead
        if(i < 0 or i > 3): continue
        for j in range(col_index - 2, col_index + 3):  # 2 cols behind, 2 cols ahead
            if(j < 0 or j > 3): continue
            if(((i == row_index - 2 or i == row_index + 2) and (j == col_index - 1 or j == col_index + 1)) or
             (((i == row_index - 1 or i == row_index + 1) and (j == col_index - 2 or j == col_index + 2)))): continue
            if(i == row_index - 2 or i == row_index + 2 or j == col_index - 2 or j == col_index + 2):
                middle_i = int((row_index + i)/2)
                middle_j = int((col_index + j)/2)
if(self.board.boards[self.player][color_side][middle_i][middle_j] != ' ‘): continue
            if(self.board.boards[self.player][color_side][i][j] == ' ‘): options.append([i, j])
    return options
```

Search Problem Formulation

Operator Preconditions (continuation):

“legalAgressiveMoves” returns which pieces can perform an aggressive move, for a given movement and board colour.

```
def legalAgressiveMoves(self, offset, other_color, piece, other_piece):
    options1 = []
    options2 = []
        for row in range(4):
            for col in range(4):
                if(self.board.boards[0][other_color][row][col] == piece):
                    if(self.verifyDirection(0, other_color, row, col, offset, piece, other_piece)):
                        options1.append([row, col])
                if(self.board.boards[1][other_color][row][col] == piece):
                    if(self.verifyDirection(1, other_color, row, col, offset, piece, other_piece)):
                        options2.append([row, col])
    return [options1, options2]
```

Search Problem Formulation

Operator Effects:

```
def updateBoard(self, passive_piece, agressive_piece, offset, piece, other_piece):
    self.board.boards[passive_piece[0]][passive_piece[1]][passive_piece[2]][passive_piece[3]] = ' ‘
    self.board.boards[passive_piece[0]][passive_piece[1]][passive_piece[2] + offset[0]][passive_piece[3] + offset[1]] = piece
    self.board.boards[agressive_piece[0]][agressive_piece[1]][agressive_piece[2]][agressive_piece[3]] = ' ‘

    v_dir, h_dir = 0
    if(offset[0] != 0): v_dir = int(offset[0] / abs(offset[0]))
    if(offset[1] != 0): h_dir = int(offset[1] / abs(offset[1]))
    n_iter = max(abs(offset[0]), abs(offset[1]))
    pushing = False
    for i in range(1, n_iter + 1):
        if(self.board.boards[agressive_piece[0]][agressive_piece[1]][agressive_piece[2] + i*v_dir][agressive_piece[3] + i*h_dir] == other_piece): pushing = True
        if(i == n_iter): self.board.boards[agressive_piece[0]][agressive_piece[1]][agressive_piece[2] + i*v_dir][agressive_piece[3] + i*h_dir] = piece
        else: self.board.boards[agressive_piece[0]][agressive_piece[1]][agressive_piece[2] + i*v_dir][agressive_piece[3] + i*h_dir] = ' '
        if(pushing):  # if there's enemy piece to be pushed
            if(agressive_piece[2] + offset[0] + v_dir in [0, 1, 2, 3] and agressive_piece[3] + offset[1] + h_dir in [0, 1, 2, 3]):
                self.board.boards[agressive_piece[0]][agressive_piece[1]][agressive_piece[2] + offset[0] + v_dir][agressive_piece[3] + offset[1] + h_dir] = other_piece
```

Search Problem Formulation

Operator Costs:

1

Evaluation Function:

- 1. countNumPieces: Evaluate the number of pieces on each board (More White Pieces -> Positive Number else -> Negative)
- 2. calcDiffNumPieces: Calculate the value of the pieces of player, used to evaluate the game when the computer is on easy mode
- 3. calcPoints: Calculate Board*Abs(Board) Sum of all Boards.
- Further explained in slide 13

Implemented Algorithms

Considering our projects needs, we used the Minimax algorithm to evaluate the plays When in computer vs computer mode, minimizing or maximazing the total score of a player, given a list of the legal moves.

We also applied the alpha-beta prunning to reduce the amount of states visited.

```python
def minimax(self, board, repeated, depth_size, depth, alpha, beta, maximizing, turn, piece, other_piece):

    if(self.mode == 2):
        difficulty = self.difficulty
    elif(self.mode == 3):
        if(turn == 1): # black player
            difficulty = self.difficultyBlack
        else:
            difficulty = self.difficultyWhite
    else:
        exit()

    if depth == 0:
        return [board.calcPoints(turn, difficulty, self), None, None, None]

    moves_sorted = self.getLegalMoves(board, repeated, turn)
    turn = self.switch_01(turn) # change player pov

    if maximizing:      # white to play (wants to maximize score)
        best = [ sys.maxsize, None, None, None]
        for move in moves_sorted:
            updated_board = Board()
            updated_board.boards = board.copyBoard()
            self.updateBoard(move[0], move[1], move[2], piece, other_piece, updated_board)
            repeated.append(updated_board)
            score = self.minimax(updated_board, repeated, depth_size, depth-1,alpha,beta,False,turn, other_piece, piece)
            repeated.pop()
            if(score[0] > best[0] or (score[0] == best[0] and random.randrange(0,6) == 3)): # score value > best value
                if(depth == depth_size):
                    best = [score[0], move[0], move[1], move[2]]
                else:
                    best[0] = score[0]
            alpha = max(alpha,best[0])
            if(alpha >= beta):
                break
```

Figure 2 - part of minimax

Experimental Implementations

During the Project's implementation, we encountered some issues regarding the code efficiency, mostly When we implemented the Minimax algorithm. Both functions to get all legal moves and to calculate the value of a board after each move were quite slow when called so many times by Minimax. When using a depth higher than 3 the moves took a really long time (5-10min) which is why we decided to not make these depths available.

In order to improve the speed of minimax we tried to sort the list of legal moves using the sorted() function by associating it with a function to calculate the points of a move as the key for the sorted function. However, this took a large amout of elapsed time (e.g for a player turn, it sometimes took about 15 to 25 seconds to do a move). So we decided against it, by choosing another approach (the function was left commented in our code under the name sortMoves).

Estatistcs

Graphic for the average execution time for levels: super easy, easy and medium, in seconds.

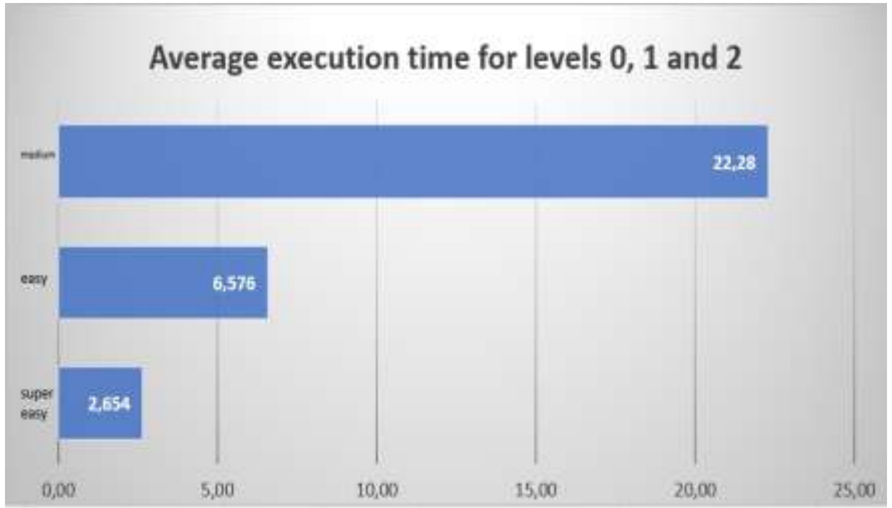


Figure 3

Graphic for the execution time for each round during a game on hard mode

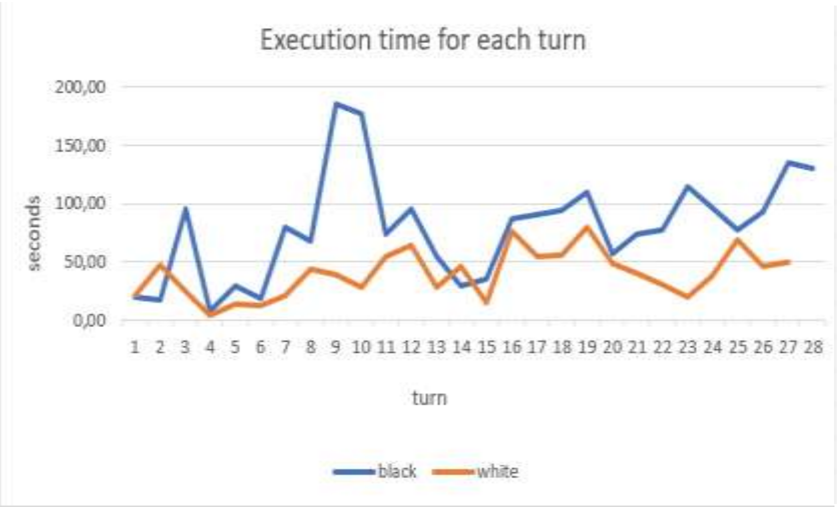


Figure 4

Graphic for the execution time for each round during a game on dynamic hard mode

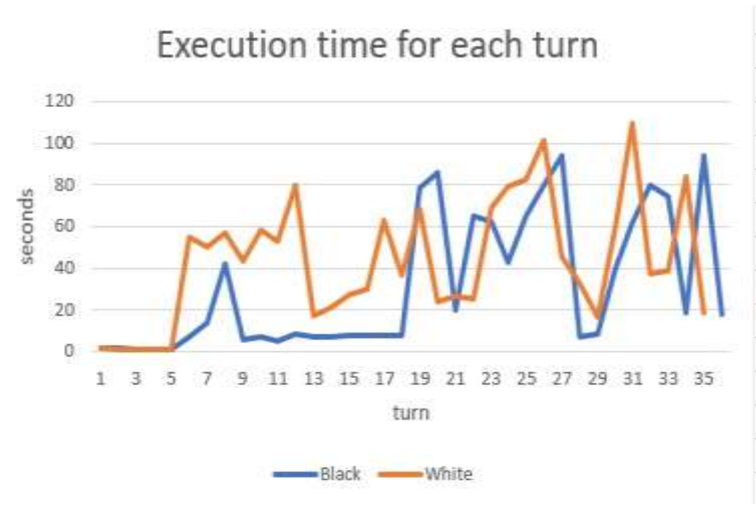


Figure 5

Implemented Work

- The language of choice is Python.
- The code is in a single file, separated in two classes: Board and GameLogic.
- PvP mode is fully functional with a clean UI. A player can also ask for a hint from the CPU, which does this using a Medium difficult.
- PvCPU is fully functional, you can also choose the difficulty of the computer. A player can also ask for a hint from the CPU, which uses the same difficulty it is playing against.
- CPUvCPU is also fully functional, being able to choose between difficulties for each CPU.
- A set of basic instructions is also available in-game.

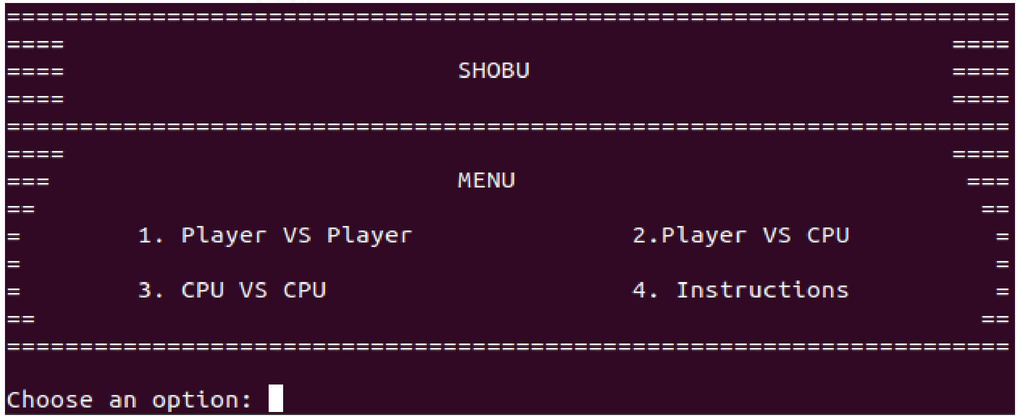


Figure 6 – Main Menu

Implemented Work

- We implemented 5 different difficulties ranging from 0-5 as in Figure 4. Difficulties 1-5 use all the same evaluation function called calcPoints (discussed in the next slide) and all use minimax with alpha-beta cuts
- Difficulty 0 acts randomly.
- Difficulty 1 uses depth = 1
- Difficulty 2 uses depth = 2
- Difficulty 3 uses depth = 3
- Difficulty 3 uses depth = 2 for the first 5 moves it makes and then changes to depth = 3, this is in order to make it faster since the first moves are not so important and yet they are the ones who take the longest due to their being so many options

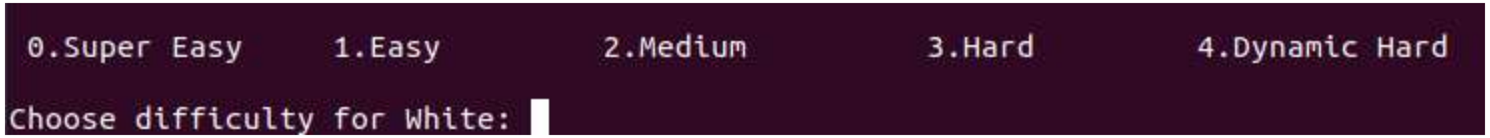


Figure 7 – CPU difficulties

Implemented Work

Our evaluation function is called calcPoints and takes into account the number of pieces on each board and adds or substracts points from the returned value acording to what player has the advantage. If value > 0 → White has advantage, Else If value < 0 → Black has advantage, the more distant from 0 the value is the more advantage a player has.

The value also takes into account which player's turn it is, giving small advantage to the player whose turn is.

A value is calculated in each board independently and then they are all joined in the end, however this is not done by simply adding the values. In order to discourage minimax from ignoring a given board and only looking out for the general value of all boards, we first multiply the value of each board by its absolute and only then add them together. As a consequence, boards with really extreme values (far from 0) will weigh a lot more in the returned value, which is important since in shobu a player only needs to win or lose in one board for the game to end.