IART
Assignment 1 Checkpoint

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.

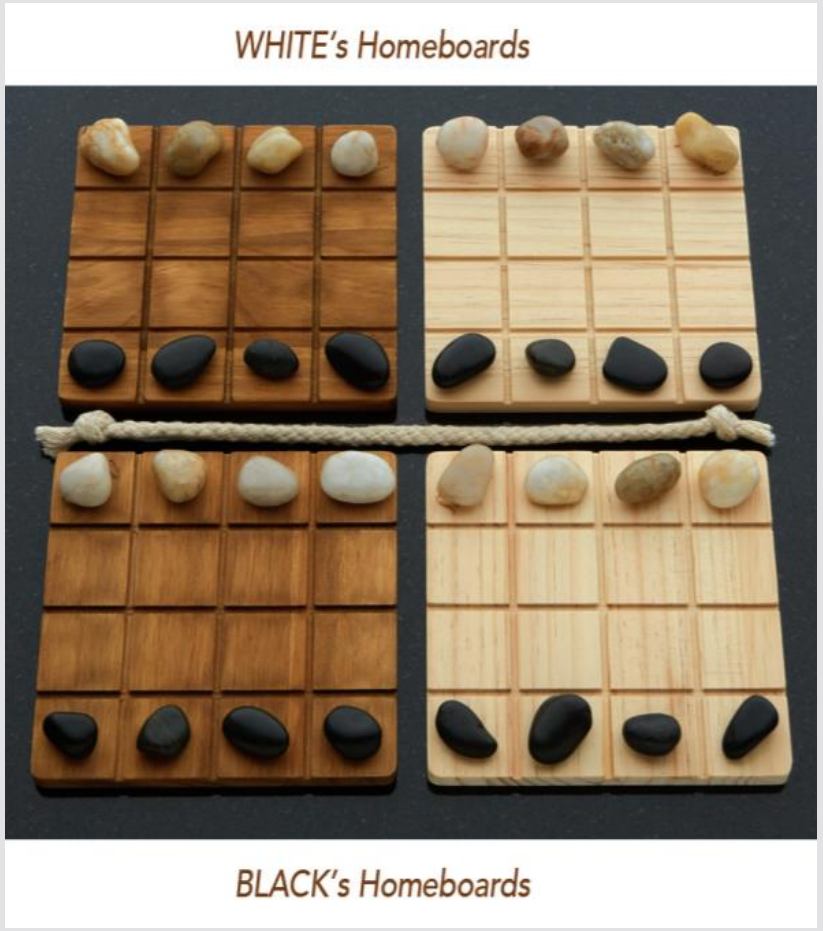

Fig. 1 – Shobu Initial Boards

References

https://www.smirkandlaughter.com/shobu

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 Preconditions (continuation):

“verifyDirection” checks if a given movement is feasible, for a given piece.

```
def verifyDirection(self, player_side, color_side, row, col, offset, piece, other_piece):
    if(row + offset[0] not in [0, 1, 2, 3] or col + offset[1] not in [0, 1, 2, 3]): return False
    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[player_side][color_side][row + i*v_dir][col + i*h_dir] == piece): return False
        if(self.board.boards[player_side][color_side][row + i*v_dir][col + i*h_dir] == other_piece): pushing = True
        if(pushing):
            if(row + (i+1)*v_dir in [0, 1, 2, 3] and col + (i+1)*h_dir in [0, 1, 2, 3]):
                if(self.board.boards[player_side][color_side][row + (i+1)*v_dir][col + (i+1)*h_dir] != " "): return False
    return True
```

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. Get List of Legal Passive Moves
2. For Each Legal Passive Move -> passMove
   3. Get List of Legal Agressive Moves
   4. For Each Legal Aggressive Move -> aggroMove
      5. scoreNumPieces[] = Evaluate Num Pieces On Each Board (More White Pieces -> Greater Number (positive), else -> Smaller Number (negative))
      6. For Each scoreNumPieces -> scoreNumPiece
         6.1. If scoreNumPiece is positive -> Increase score based on how few black pieces are in that board
         6.2. Else If scoreNumPiece is negative -> Decrease score based on how few white pieces are in that board
         6.3. Else -> If White's turn -> Increase score based on how many pieces are in that board, else -> Decrease
      7. totalScore = Calculate Quadratic Sum of scoreNumDPieces on each Board
      8. Save Tuple -> (totalScore, passMove, aggroMove)

Implemented Work

The language of choice is Python.

Currently, the code is in a single file, separated in two classes: Board and GameLogic.

Most of the game is already coded. PvP mode is fully functional with a clean UI. The game functions are being extended to allow PvC and CvC modes at the same time Minimax is being implemented.