Lund University

Artificial Intelligence

Assignment 1 - Adversarial Search

Cullen de Erquiaga Magdalena Itziar

Mutz Matias

Group 14

Lund University

Exchange Students

LUND UNIVERSITY

LTH

FACULTY OF ENGINEERING

February 6, 2025

Contents

1. Alpha-Beta algorithm

1.1 Minimax function

The alpha-beta algorithm is found in the minimax method, located in the line 90 up to 129. We had started the assignment trying to implement the algorithm in three methods but opted to change the design in order to organize the code more clearly. We designed a global method, which is used in the **play_game(self)** method when the *self.current_player* is not the *human_player*, see figure 1.1. It is called **minimax(self, board, depth, alpha, beta, maximizing_player, color)**, see figure 1.2, and it controls the amount of valid_moves left for the AI player to take while ensuring that the game has not ended. If the recursive function reaches **depth == 0**, indicates that it finished going through the possible outcome of the game with that move. The board state is evaluated through **heuristic(self, board, player)** and returns the score to see which is chosen by the AI player.

```
print("Invalid input")
    self.apply_move(self.board, row, col, self.current_player)
else:
    print("AI thinking...")
    _, move = self.minimax(self.board, 4, -math.inf, math.inf, True, self.current_player)
    if move:
        print(f"AI plays: {move}")
        self.apply_move(self.board, move[0], move[1], self.current_player)

self.current_player = WHITE if self.current_player == BLACK else BLACK

```

Figure 1.1: Fragment of the play_game(self) function

```
def minimax(self, board, depth, alpha, beta, maximizing_player, player):
        if depth == 0:
            return self.heuristic(board, player), None

        valid_moves = self.get_valid_moves(board, player)
        if not valid_moves:
            opponent = WHITE if player == BLACK else BLACK
            if not self.get_valid_moves(board, opponent):  # If opponent also has no moves, return heuristic
                return self.heuristic(board, player), None
            return self.minimax(board, depth - 1, alpha, beta, not maximizing_player, opponent)

        best_move = None
        if maximizing_player:
            max_eval = -math.inf
            for move in valid_moves:
                new_board = board.copy()
                self.apply_move(new_board, move[0], move[1], player)
                eval, _ = self.minimax(new_board, depth - 1, alpha, beta, False,
                                       WHITE if player == BLACK else BLACK)
                if eval > max_eval:
                    max_eval = eval
                    best_move = move
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return max_eval, best_move
        else:
            min_eval = math.inf
            for move in valid_moves:
                new_board = board.copy()
                self.apply_move(new_board, move[0], move[1], player)
                eval, _ = self.minimax(new_board, depth - 1, alpha, beta, True,
                                       WHITE if player == BLACK else BLACK)
                if eval < min_eval:
                    min_eval = eval
                    best_move = move
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return min_eval, best_move
```

Figure 1.2: Minimax function

1.2 Heuristic

Regarding the heuristic function, located in the line 71 to 88, we used a grid-based scoreboard to estimate the likelihood of winning based on the position of the disk, see figure 1.3. This is used in the **heuristic(self, board, player)** to determine which move is best for the AI player. This grid gives priority to the corners, as once you reach a corner, there is a higher chance of winning. While positions that give away the corner to the opponent are negatively valued, so they are not chosen unless there is no other option. Furthermore, it also analyzes the number of possible moves for each player, this adds a mobility score factor in the decision, prioritizing moves that maximize future opportunities for the AI, see in figure 1.4. We found that increasing this factor-weighted difference enhances the AI's ability to win the game, demonstrating that prioritizing the number of moves allowed for the human_player increases the AI's chances of winning.

```python
# Grid score
SCORES = [
    [100, -20, 10,  5,  5, 10, -20, 100],
    [-20, -50, -2, -2, -2, -2, -50, -20],
    [ 10,  -2,  5,  1,  1,  5,  -2,  10],
    [  5,  -2,  1,  2,  2,  1,  -2,   5],
    [  5,  -2,  1,  2,  2,  1,  -2,   5],
    [ 10,  -2,  5,  1,  1,  5,  -2,  10],
    [-20, -50, -2, -2, -2, -2, -50, -20],
    [100, -20, 10,  5,  5, 10, -20, 100]
]
```

Figure 1.3: Grid-based scoreboard

```python
def heuristic(self, board, player):
        opponent = BLACK if player == WHITE else WHITE
        score = 0

        # Positional values from SCORES matrix
        for i in range(8):
            for j in range(8):
                if board[i][j] == BLACK:
                    score += SCORES[i][j]
                elif board[i][j] == WHITE:
                    score -= SCORES[i][j]

        # Number of possible moves for both players
        player_moves = len(self.get_valid_moves(board, player))
        opponent_moves = len(self.get_valid_moves(board, opponent))
        score += (player_moves - opponent_moves) * 10 # Weighted difference

        return score if player == BLACK else -score
```

Figure 1.4: Heuristic function

We found this design to be the most logical, as it incorporates the algorithm we learned in class and uses a static matrix of position priorities, which is beneficial for the AI player because it increases its probability of winning the game. Winning a corner is much more valuable than winning a side, and it also prevents the worst positions that give the corner to its opponent. While also considering the number of options we leave the opponent with in comparison to us.

1.3 Observations

We observed that increasing the depth of the tree while assigning a high weighted value to the mobility score leads to worse results. This is most likely due to prioritizing the mobility score over the board positioning negatively impacting the performance of AI. Furthermore, looking too many steps ahead is very inefficient and causes the algorithm to fail since human actions are not 100% predictable. Therefore, increasing the depth more than five, the mobility score weight should drop.

1.4 Testing

The way we tested the Othello game was by playing against an online Othello game (hewgill.com/othello) to see if we could win against the normal difficulty mode (see figure 1.5). The black player was our implementation, and the white player was the online game.

We started by playing in our code as the white player (simulating the online game's moves), and our implementation made the first move, which we then played in the online game. After that, we took the online game's response and played it in our implementation. We repeated this process until the game was finished.

At first, we always lost against the online game (see figure 1.6). But with the final implementation, we started winning and becoming more competitive (see figure 1.7).

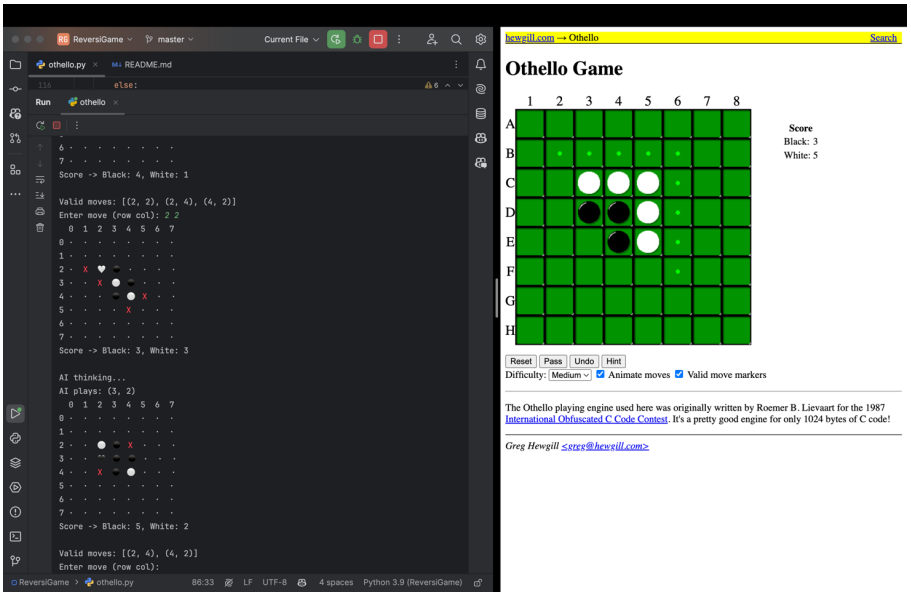


Figure 1.5: Starting to play a game between our implementation and the online game

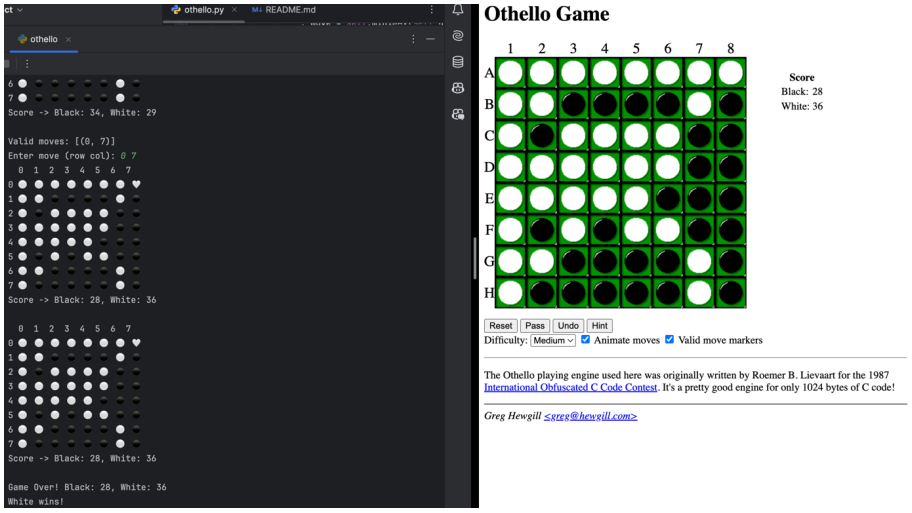


Figure 1.6: Losing a game with an old version

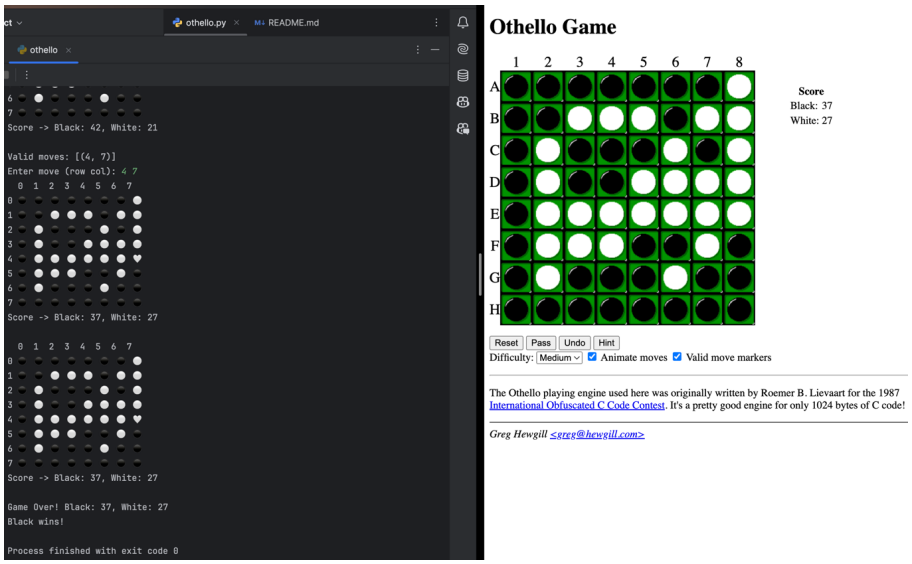


Figure 1.7: Wining a game with an old version

2. How to play Othello Game

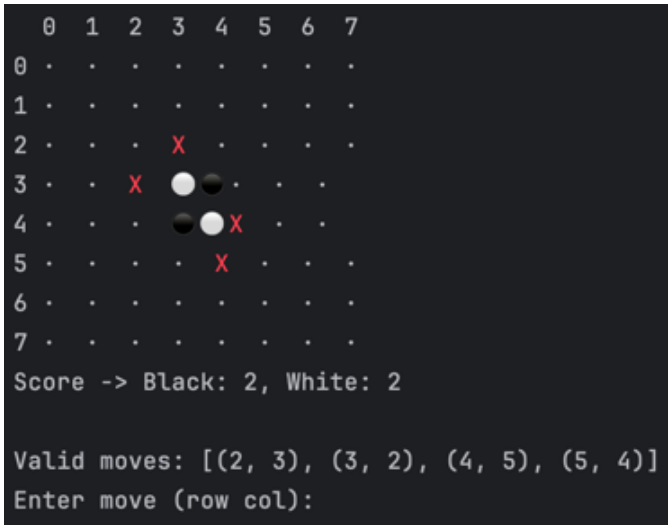
1. Execute the othello.py file.

```
ReversiGame ❱ python3 othello.py
```

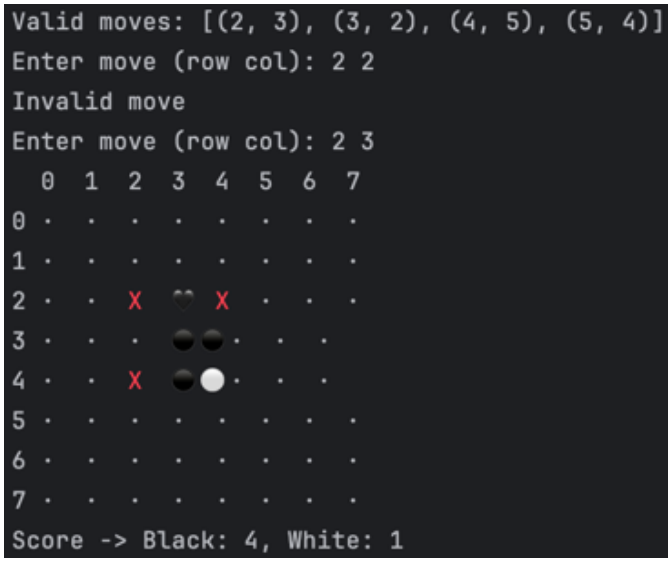
2. Selected color you want to play with, Black always starts.

```
Welcome to Othello!
Play as Black (B) or White (W)?
```

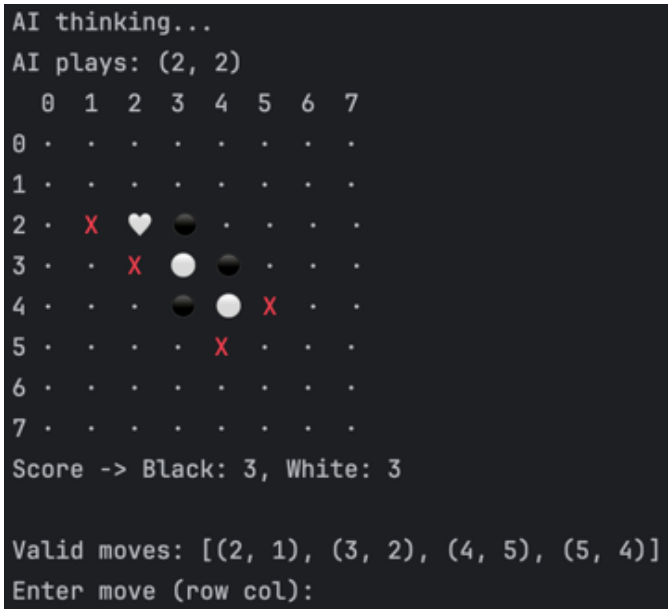
3. The game will display with red X's on the board the possible moves to take as well as listing them below.



4. To play you must select one of the valid moves by writing the row followed by the column. If the move is not valid an error message will appear and ask the player to enter a valid move again. The selected move will appear with a heart.



5. Then the AI will play and display which move was taken. The score is displayed in below the board, the valid moves and red X's will appear again.



6. Steps 3 through 5 will be repeated until the game is over.