Artificial Intelligence

Othello-game

Team (22)

1- Yossif Ekramy Ali Ibrahim (sec 5)

2-AbdalRahman Muhammad Tawfiq Abdalhamid (sec 5)

3-Rahma Yahya Mohammed Rashid (sec 5)

4-Youssif Ibrahim Ali AbdalRahman (sec 5)

5- AbdalHadi Fathi Abu Al-Anin Musa (sec 5)

. Introduction

- This project is an interactive Othello (Reversi) game developed in Python using the pygame library. The aim is to create a strategic board game experience where the player can challenge a smart AI opponent in real time.

- The AI utilizes the Minimax algorithm with Alpha-Beta Pruning to make optimal decisions, offering a competitive and dynamic gameplay. The game strictly follows Othello’s official rules, including valid move checks, automatic piece flipping, and endgame detection.

Step-by-Step Implementation

step 1: Game Logic (engine.py)
• Implemented the GameState class to manage the Othello board state dynamically.
• Represented the board as an 8x8 matrix with values indicating black, white, or empty tiles.
• Handled move validation, automatic piece flipping, and turn switching.
• Maintained a move log to support undo functionality.

Step 2: AI Logic (AI_Move.py)
• Defined the evaluate_board function to assess the board state based on piece positions and control.
• Used the Minimax algorithm with Alpha-Beta Pruning to determine the optimal move.
• Optimized search performance by pruning unpromising branches to reduce computation time.

Step 3: Main Interface (main.py)
• Created an interactive GUI using the pygame library.
• Handled user input via mouse clicks to select and place pieces.
• Highlighted valid moves for the player and triggered AI responses after each turn.
• Displayed endgame screen with winner announcement and final score.

Grid System:

-**Initializes the board with the number of rows and columns.**

-**Loads visual assets: tokens, transitions, background.**

-**Manages the game logic: legal moves, swapping tokens, score count.**

```
class Grid:
    def __init__(self, rows, columns, size, main):
        self.GAME = main
        self.y = rows
        self.x = columns
        self.size = size
        self.whitetoken = loadImages('WhiteToken.png', size)
        self.blacktoken = loadImages('BlackToken.png', size)
        self.font = pygame.font.SysFont('Arial', 20, True, False)
        self.transitionWhiteToBlack = [loadImages(f'BlackToWhite{i}.png', self.size) for i in range(1, 4)]
        self.transitionBlackToWhite = [loadImages(f'WhiteToBlack{i}.png', self.size) for i in range(1, 4)]
        self.player1Score = 0
        self.player2Score = 0
        self.bg = self.loadBackGroundImages()
        self.tokens = {}
        self.gridBg = self.createbgimg()
        self.gridLogic = self.regenGrid(self.y, self.x)
```

AI Logic:

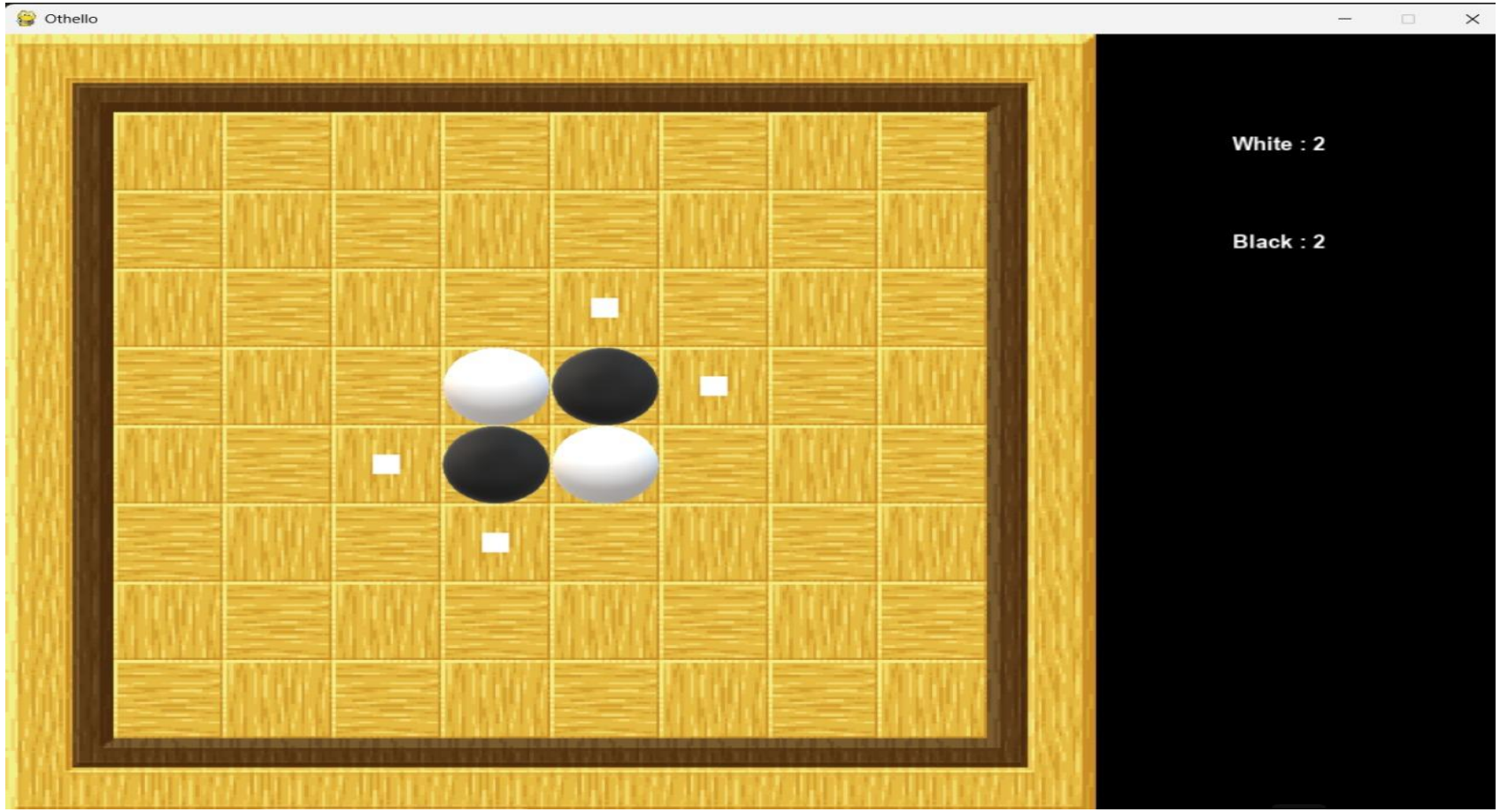
Uses Minimax with Alpha-Beta pruning.

Evaluates the board and chooses the best move based on depth.

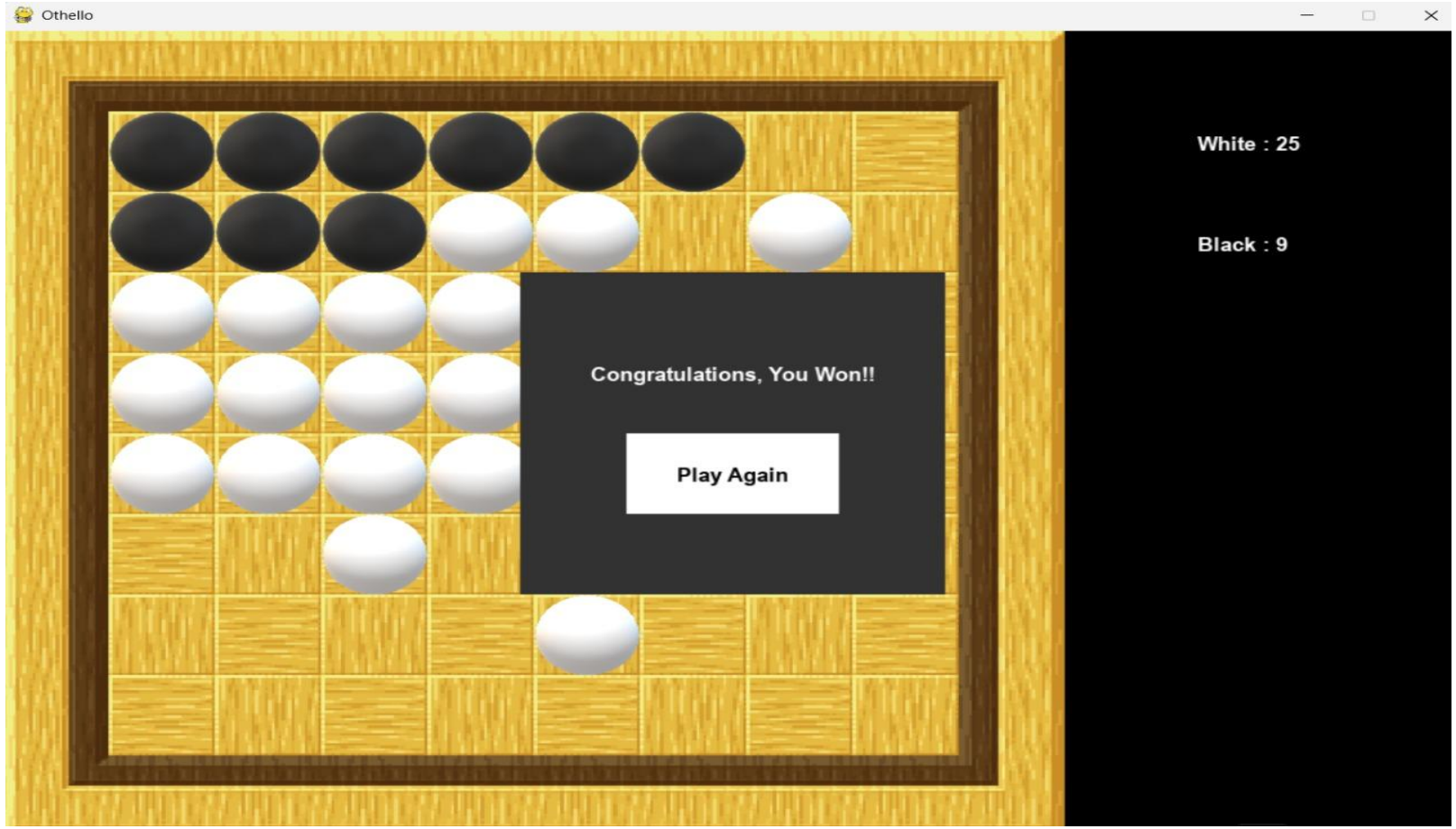
```python
class ComputerPlayer:
    def __init__(self, gridObject):
        self.grid = gridObject

    def searchFunction(self, depth, move, newGrid, player, alpha, beta):
        X, Y = move
        swappableTiles = self.grid.swappableTiles(X, Y, newGrid, player)
        newGrid[X][Y] = player
        for tile in swappableTiles:
            newGrid[tile[0]][tile[1]] = player

        bestMove, value = self.computerHard(newGrid, depth-1, alpha, beta, player *-1)

    def computerHard(self, grid, depth, alpha, beta, player):
        newGrid = copy.deepcopy(grid)
        availMoves = self.grid.findAvailMoves(newGrid, player)
        if depth == 0 or len(availMoves) == 0:
            bestMove, Score = None, self.evaluateBoard(grid, player)
            return bestMove, Score


        if player < 0:
            bestScore = -64
            bestMove = None


            for move in availMoves:
                X, Y = move
                swappableTiles = self.grid.swappableTiles(X, Y, newGrid, player)
                newGrid[X][Y] = player
                for tile in swappableTiles:
                    newGrid[tile[0]][tile[1]] = player
                bestMove, value = self.computerHard(newGrid, depth-1, alpha, beta, player *-1)
                if value > bestScore:
                    bestScore = value
                    bestMove = move

                alpha = max(alpha, bestScore)
                if beta <= alpha:
                    break
```

```python
            newGrid = copy.deepcopy(grid)
        return bestMove, bestScore
    if player > 0:
        bestScore = 64
        bestMove = None


        for move in availMoves:
            X, Y =move
            swappableTiles = self.grid.swappableTiles(X, Y, newGrid, player)
            newGrid[X][Y] = player
            for tile in swappableTiles:
                newGrid[tile[0]][tile[1]] = player

            bMove, value = self.computerHard(newGrid, depth-1, alpha, beta, player)
            if value < bestScore:
                bestScore = value
                bestMove = move
            beta = min(beta, bestScore)
            if beta <= alpha:
                break
            newGrid = copy.deepcopy(grid)
        return bestMove, bestScor
```

Main Interface:

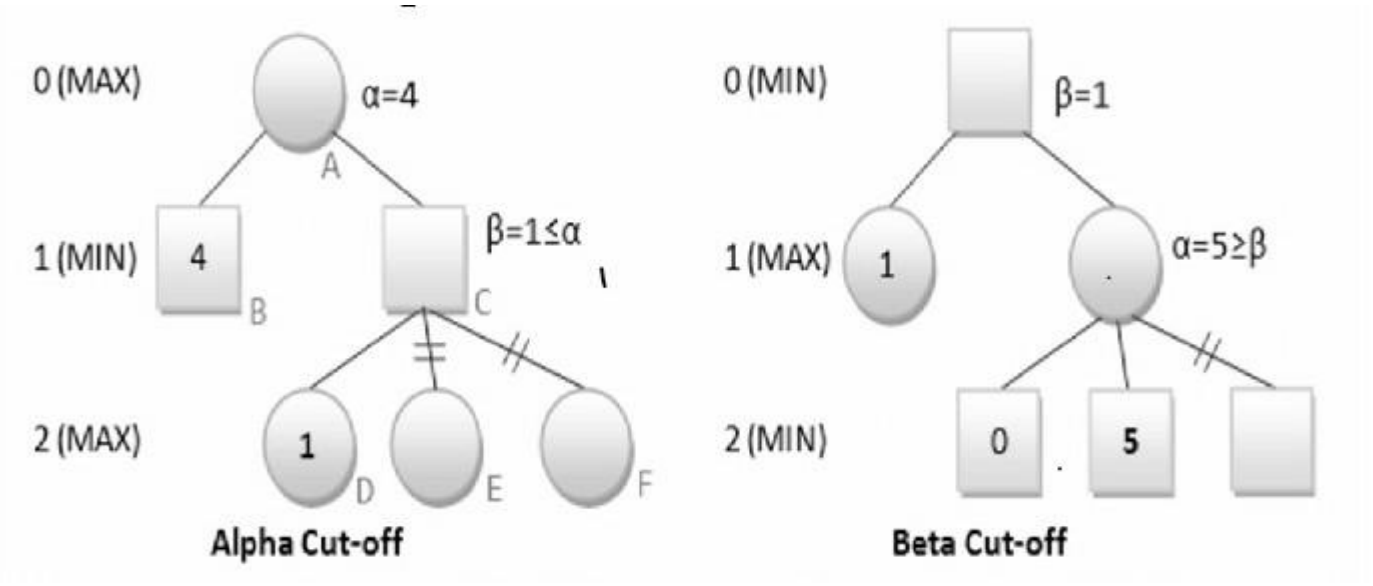


End Game Intarface:



Alpha-Beta Pruning:

- optimization of Minimax.
- Alpha: Best already explored option for the maximizer.
- Beta: Best already explored option for the minimizer.
- Prunes branches that won't affect the final decision ➞ faster AI.



Conclusion :

- **Strengths:**
  - Intelligent move selection using Minimax with Alpha-Beta Pruning.
  - Clear modular design separating game logic, AI, and GUI.
  - Smooth user interface with real-time updates and valid move indicators.
- **Challenges:**
  - Managing turn-passing logic when no valid moves are available.
  - Ensuring accurate piece flipping in all directions under complex board states.
  - Optimizing performance for deeper AI lookahead without lag