Assignment2

Connect-4

Report

Abdelrahman Amr Salah 8016
Ahmed Mahmoud Aly 8062
Youssef Mohamed Attia 8164

Overview:

our project is multi-module project consists of Solver, Game-window ,Test, tree rep. , Analyse modules as they imports solver module which contain needed functions and classes

Data Structures Used:

1. **2D NumPy Array (np.array):**
    - The board is represented as a 2D array, where each cell can hold a value representing an empty space, the player's piece, or the AI's piece.
    - Example: board.current_state is a 2D NumPy array representing the current board configuration.
2. **List:**
    - valid_columns (in get_neighbors): A list that holds the indices of columns where pieces can be dropped.
    - columns (in get_cols): A list of columns neighboring the current column, used for probabilistic moves in the ExpectiMiniMax algorithm.
    - probability_distribution (in get_cols): A list representing the probabilities associated with choosing each neighboring column.
    - cols (in many methods): Holds potential column indices to place the next piece.
3. **□ Tuples:**
    - Return values: Multiple methods return tuples, where the first element is a column index, and the second is the evaluation score (e.g., col, value).

Solver code Breakdown:

```python
class Solver:
    def __init__(self, depth=5, Ai_piece=1, player_piece=2):
        self.max_depth = depth
        self.ai_piece = Ai_piece
        self.player_piece = player_piece

    9 usages (2 dynamic)
    def solve(self, board, solver="minmax"):
        col = None
        value = None
        if solver.lower() == "minmax":
            col, value = self.MiniMax(board, depth: 0, is_maximizer: True)
        elif solver.lower() == "alphabetapruning":
            col, value = self.MiniMax_alpha_beta_pruning(board, depth: 0, -math.inf, math.inf, is_maximizer: True)
        elif solver.lower() == "ExpectMiniMax".lower():
            col, value = self.ExpectiMiniMax(board, depth: 0, is_maximizer: True)
        return col, value
```

1. **__init__ Method:**
    a. Initializes the solver with a maximum search depth (depth), the AI's piece identifier (Ai_piece), and the player's piece identifier (player_piece).
    b. self.max_depth: Defines how deep the AI will search for optimal moves.
    c. self.ai_piece and self.player_piece: Used to distinguish between the AI's and the human player's pieces on the board.
2. **solve Method:**
    a. this method selects the appropriate solving algorithm (Minimax, Alpha-Beta Pruning, or ExpectiMiniMax) and returns the best column (col) and the associated value (value) for that move.

```python
def count_fours(self, board, piece):
    count = 0
    rows, cols = board.shape
    # Count all horizontal, vertical, and diagonal sequences of 4 connected pieces
    for r in range(rows):
        for c in range(cols - 3):
            if np.all(board[r, c:c + 4] == piece):
                count += 1

    for c in range(cols):
        for r in range(rows - 3):
            if np.all(board[r:r + 4, c] == piece):
                count += 1

    for r in range(rows - 3):
        for c in range(cols - 3):
            if np.all([board[r + i, c + i] == piece for i in range(4)]):
                count += 1

    for r in range(3, rows):
        for c in range(cols - 3):
            if np.all([board[r - i, c + i] == piece for i in range(4)]):
                count += 1

    return count
```

3. count_fours Method:

a. Counts the number of "four-in-a-row" sequences on the board for a given piece.

b. This method scans the board horizontally, vertically, and diagonally for four connected pieces that match the given piece.

```
def MiniMax(self, board, depth, is_maximizer=True):
    if depth >= self.max_depth or board.available_places == 0:
        return self.evaluate_board(board)

    if is_maximizer:
        value = -math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.ai_piece)
            _, score = self.MiniMax(board, depth + 1, is_maximizer: False)
            board.remove_piece(col)  # Undo move

            if score > value:
                value = score
                best_col = col
        return best_col, value
    else:
        value = math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.player_piece)
            _, score = self.MiniMax(board, depth + 1, is_maximizer: True)
            board.remove_piece(col)  # Undo move

            if score < value:
                value = score
                best_col = col
        return best_col, value
```

4. MiniMax Method:

a. Implements the basic Minimax algorithm. It recursively explores the game tree, alternating between maximizing (AI) and minimizing (player) at each depth level.

b. If the maximum depth is reached or there are no available places, the algorithm evaluates the board's state.

c. Maximizer's Turn (AI): Looks for the move that maximizes the score.

d. Minimizer's Turn (Player): Looks for the move that minimizes the score.

```python
def MiniMax_alpha_beta_pruning(self, board, depth, alpha, beta, is_maximizer=True):
    if depth >= self.max_depth or board.available_places == 0:
        return self.evaluate_board(board)

    if is_maximizer:
        value = -math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.ai_piece)
            _, score = self.MiniMax_alpha_beta_pruning(board, depth + 1, alpha, beta, is_maximizer: False)
            board.remove_piece(col)  # Undo move

            if score > value:
                value = score
                best_col = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return best_col, value
    else:
        value = math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.player_piece)
            _, score = self.MiniMax_alpha_beta_pruning(board, depth + 1, alpha, beta, is_maximizer: True)
            board.remove_piece(col)  # Undo move

            if score < value:
                value = score
                best_col = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return best_col, value
```

5. MiniMax_alpha_beta_pruning Method:

a. Similar to Minimax but with alpha-beta pruning, which cuts off branches in the search tree that are not likely to affect the final decision. This speeds up the process by reducing the number of nodes evaluated.

b. Alpha: Represents the best score that the maximizer can guarantee.

c. Beta: Represents the best score that the minimizer can guarantee.

d. Pruning: If at any point alpha is greater than or equal to beta, further exploration of that branch is stopped (pruned).

```python
def ExpectiMiniMax(self, board, depth, is_maximizer=True):
    if depth >= self.max_depth or board.available_places == 0:
        return self.evaluate_board(board)

    if is_maximizer:
        value = -math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.ai_piece)
            expected_value = 0
            neighboring_cols, prob = self.get_cols(board, col)

            for i, new_col in enumerate(neighboring_cols):
                if 0 <= new_col < board.cols and board.first_empty_tile(new_col) is not None:
                    board.add_piece(new_col, self.ai_piece)
                    _, score = self.ExpectiMiniMax(board, depth + 1, is_maximizer: False)
                    board.remove_piece(new_col)  # Undo move
                    expected_value += prob[i] * score

            board.remove_piece(col)  # Undo move

            if expected_value > value:
                value = expected_value
                best_col = col
        return best_col, value
```

```python
    else:
        value = math.inf
        best_col = None
        cols = self.get_neighbors(board)

        for col in cols:
            board.add_piece(col, self.player_piece)
            expected_value = 0
            neighboring_cols, prob = self.get_cols(board, col)

            for i, new_col in enumerate(neighboring_cols):
                if 0 <= new_col < board.cols and board.first_empty_tile(new_col) is not None:
                    board.add_piece(new_col, self.player_piece)
                    _, score = self.ExpectiMiniMax(board, depth + 1, is_maximizer: True)
                    board.remove_piece(new_col)
                    expected_value += prob[i] * score

            board.remove_piece(col)

            if expected_value < value:
                value = expected_value
                best_col = col
        return best_col, value
```

6. ExpectiMiniMax Method:

a. An extension of the Minimax algorithm incorporating probabilistic outcomes, which is useful in games with elements of chance.

b. The AI considers not only the immediate moves but also the potential responses of the player and their probabilities.

c. The method calculates an "expected value" instead of a fixed best or worst outcome

```
def evaluate_board(self, board):
    if board.available_places == 0:
        player_4s = self.count_fours(board.current_state, self.player_piece)
        ai_4s = self.count_fours(board.current_state, self.ai_piece)
        if player_4s > ai_4s:
            return (None, -math.inf)
        elif ai_4s > player_4s:
            return (None, math.inf)
        else:
            return (None, 0)
    else:
        return (None, board.calculate_score(self.ai_piece))

6 usages
def get_neighbors(self, board):
    valid_columns = [col for col in range(board.cols) if board.first_empty_tile(col) is not None]
    random.shuffle(valid_columns)
    return valid_columns
```

```
2 usages
def get_cols(self, board, col):
    if col == 0:
        probability_distribution = [0.6, 0.4]
        columns = [col, col + 1]
    elif col == board.cols - 1:
        probability_distribution = [0.4, 0.6]
        columns = [col - 1, col]
    else:
        probability_distribution = [0.2, 0.6, 0.2]
        columns = [col - 1, col, col + 1]
    return columns, probability_distribution
```

7. **evaluate_board Method:**
    a. Evaluates the board state to provide a score that reflects the AI's advantage.
    b. If the board is full (board.available_places == 0), it counts the "four-in-a-row" sequences for both players and returns the score accordingly.
    c. If not full, it calculates the score based on the AI's piece position on the board.
8. **get_neighbors Method:**
    a. Returns a list of valid columns where a piece can be placed, shuffled to add randomness.
9. **get_cols Method:**
    a. Provides neighboring columns and their associated probabilities, used in the ExpectiMiniMax method to model uncertainty.

Sample runs:

| 1.0 | 2.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 2.0 | 2.0 | 2.0 | 0.0 |

| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 1.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 1.0 | 2.0 | 2.0 | 0.0 |

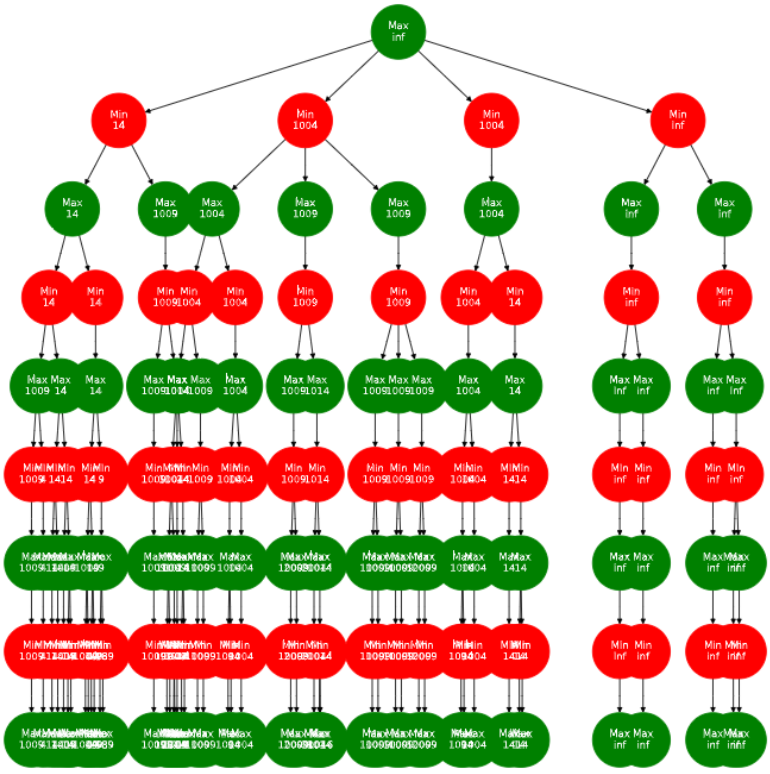
| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 2.0 | 2.0 |

| 1.0 | 1.0 | 1.0 | 1.0 | 2.0 | 1.0 | 1.0 |

---

1 2 3 4 5 6 7

Time Taken 0.0030388832092285156



| 1.0 | 2.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 2.0 | 0.0 | 0.0 | 0.0 |

| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 0.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 1.0 | 2.0 | 0.0 | 0.0 |

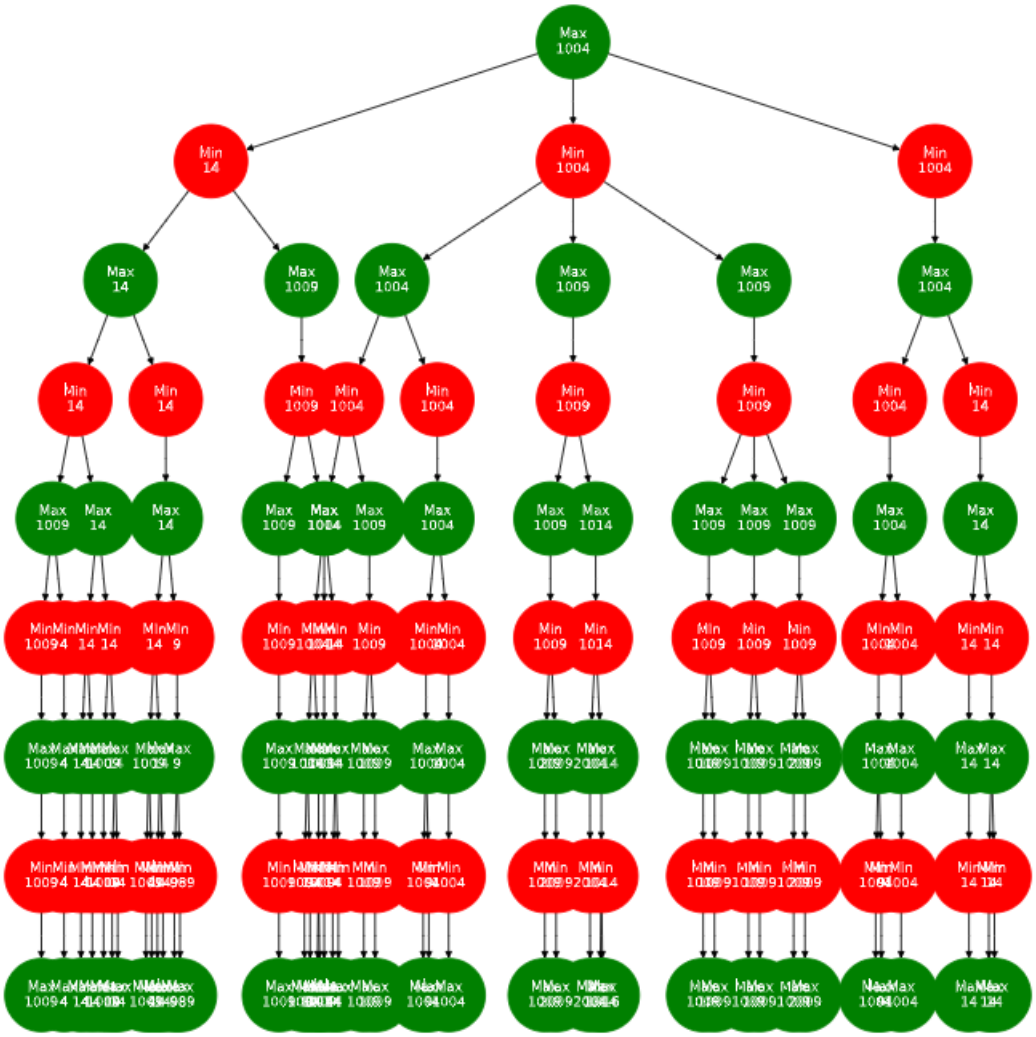
| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 0.0 | 0.0 |

| 1.0 | 1.0 | 1.0 | 1.0 | 2.0 | 0.0 | 0.0 |

---

1 2 3 4 5 6 7

Time Taken 0.12709999084472656



| 1.0 | 2.0 | 1.0 | 1.0 | 1.0 | 0.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 2.0 | 2.0 | 0.0 | 0.0 |

| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 0.0 | 0.0 |

| 2.0 | 1.0 | 1.0 | 1.0 | 2.0 | 0.0 | 0.0 |

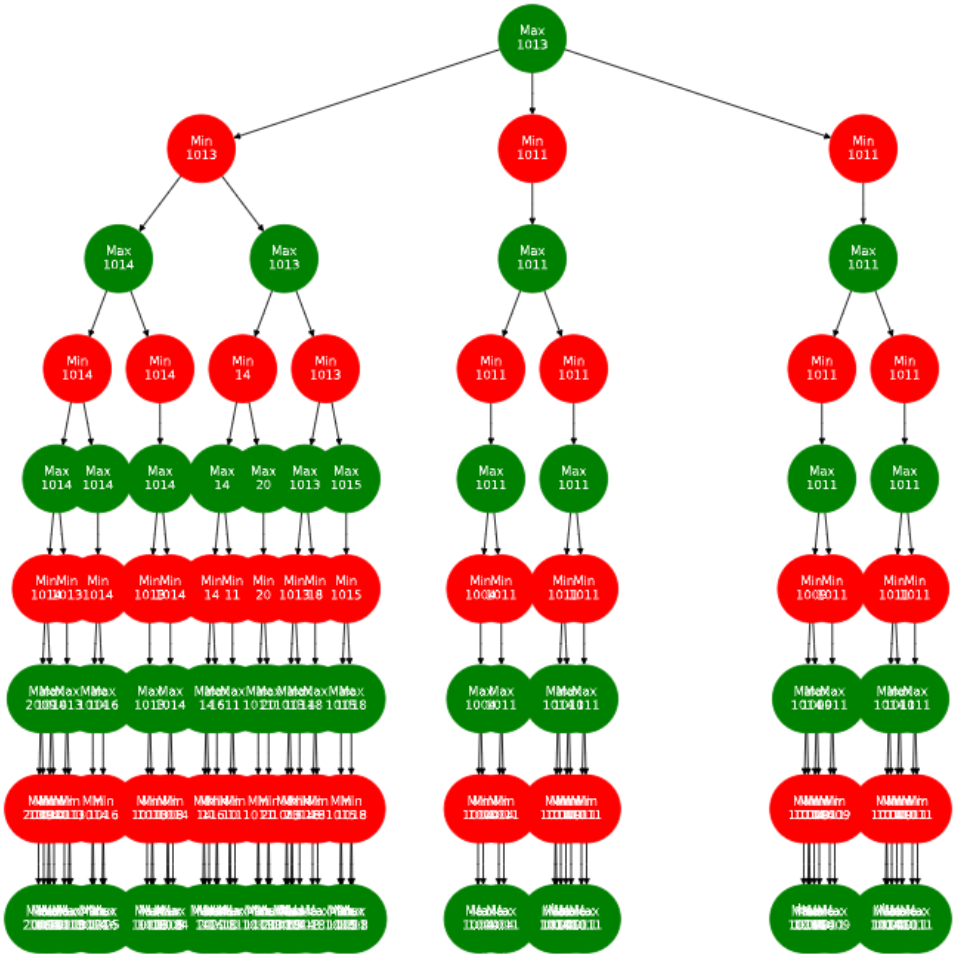
| 1.0 | 2.0 | 2.0 | 2.0 | 1.0 | 0.0 | 0.0 |

| 1.0 | 1.0 | 1.0 | 1.0 | 2.0 | 2.0 | 0.0 |

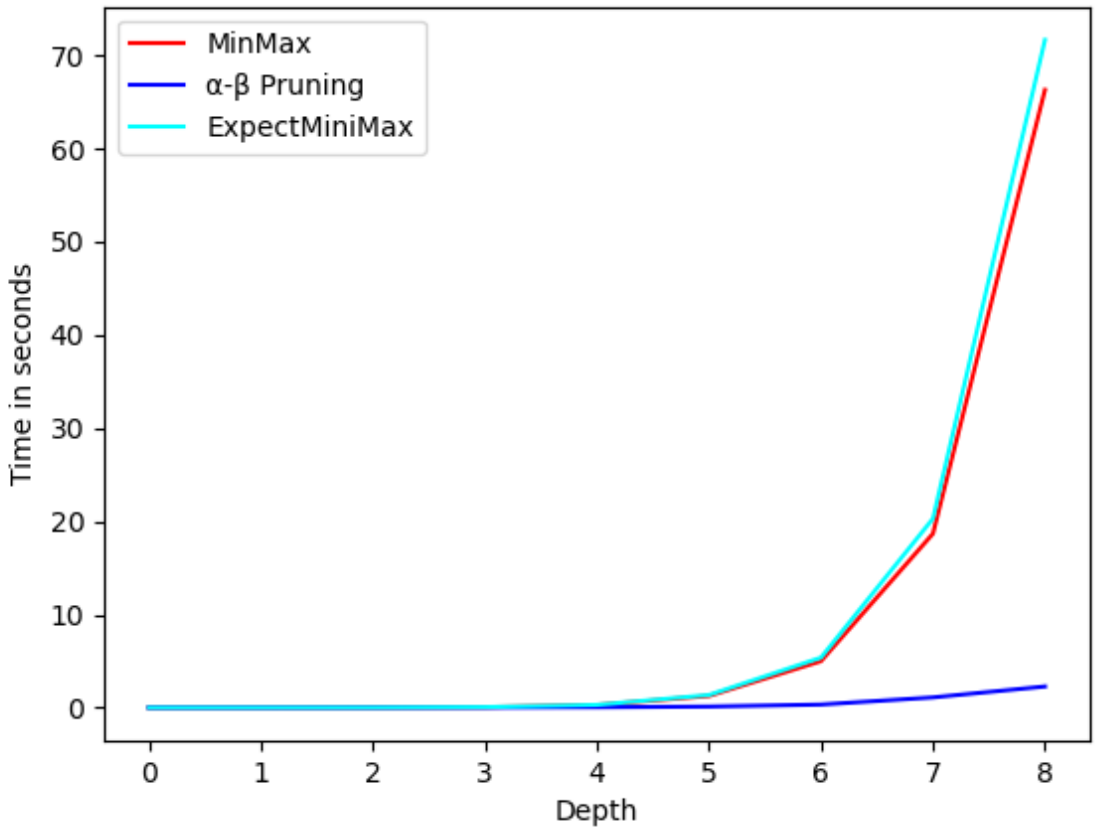
---

1 2 3 4 5 6 7

Time Taken 0.023257970809936523



Comparison:


MinMax
α-β Pruning
ExpectMiniMax
Time in seconds
Depth
0
10
20
30
40
50
60
70
0
1
2
3
4
5
6
7
8