1. Compilation and Execution

The program does not need to be compiled
Run the following command to execute the program:

```
python3 main.py
```

Note that the program will only run on one of the lab machines, and will not run on Otter.

Constants for the game must be modified in the Board.py File.

2. Minimax and alpha beta

the minimax algorithm with alpha beta pruning is implement in AdversarialSearch.py:42 as the following function:

```
def minimax(self, state, depth, alpha, beta, maximizingPlayer):
```

Parameters

- state (State object): The current game state.
- depth (int): The depth of the game tree to explore. A depth of 0 will return the evaluation of the current state.
- alpha (int): The best score that the maximizing player can guarantee at this level or above.
- beta (int): The best score that the minimizing player can guarantee at this level or above.
- maximizingPlayer (bool): True if the current player is maximizing, False if minimizing.

Full Function

```
        if depth == 0 or gameOver(state):
            return self.evaluateBoard(state)

        if maximizingPlayer:
            maxEval = float('-inf')
            for move in state.availableMoves:
                eval = self.minimax(self.updateState(deepcopy(state), move), depth - 1, alpha, beta,
False)
                maxEval = max(maxEval, eval)

                # prunning
                alpha = max(alpha, eval)
                if beta <= alpha:
                    break
            return maxEval

        else:
            minEval = float('inf')
            for move in state.availableMoves:
                eval = self.minimax(self.updateState(deepcopy(state), move), depth - 1, alpha, beta,
True)
                minEval = min(minEval, eval)

                # pruning
                beta = min(beta, eval)
                if beta <= alpha:
                    break
            return minEval
```

The minimax function returns the maximum score for a given move. It is called by this function in AdversarialSearch.py:24

```
def findOptimalMove(self, board, top):
```

The above function is called by the AgentPlayer.py in order to find the best move to play. findOptimalMove will call minimax on all available moves and return the move with the highest evaluation.

Full Function:

```
    def findOptimalMove(self, board, top):
        currentState = State(board, self.marble, top)

        bestMove = -1
        bestValue = float('-inf')
```

```
# Call minimax function on each available move
for col in currentState.availableMoves:
    newState = self.updateState(deepcopy(currentState), col)
    moveValue = self.minimax(newState, self.depth, float('-inf'), float('inf'), False)

    if moveValue > bestValue:
        bestValue = moveValue
        bestMove = col

# return the move with the highest evaluation
return bestMove
```

3. Evaluation Function

The evaluation function is defined in AdversarialSearch.py:102

```
def evaluateBoard(self, state):
```

It takes the current state of the board and evaluates it based on several qualities.

First it checks if either player has won the game. It will return 100 if the Agent player has won and -100 if the adversary has won. 0 is returned for a draw.

If the game is not yet over, it will check for threats. Threats are when either player has i marbles lined up with *goal - i* available spaces. Eg: for i=3 '_'XXX or XXX'_'. Or for i =1 '_'_'_'X.

Threats are counted diagonally, horizontally and vertically for both players. I weighed a threat of 3 in a row as 7 points, a threat of 2 in a row as 5 points, and a threat of 1 in a row as 3 points. I kept these numbers fairly low so as to not interfere with a game over score. Threat values are subtracted for adversary threats and added for player threats.

Center column: It is more advantageous for the AI to play in one of the center columns (because you can build your row on either side), so I assigned a value of 1 to the empty center columns for the start of the game. It is only 1 because it just needs to be greater than 0 (as the edge columns will be evaluated to 0 at the start of the game), but I also don't want this move to be prioritized over any other move (as described in the previous section).

4. Any Known Bugs

Limitation of Minimax

My AI agent has one flaw (that I could find).

Situation: There are two moves that could lead to a win (and have the same evaluation score), but one move will lead to a win in 2 moves, while the other will lead to a win in 1 move. The AI won't pick the option with that is closests to the current state. It will pick the option of the column closest to the left side of the connect 4 board.

My understanding is that the minimax algorithm is not inherently designed to catch this, but it would be nice to implement a fix. The minimax algorithm could return the depth that an answer was found at, or the evaluation function could account for current depth if there is a winning state.

I did not implement any fixes for this flaw in my program, as winning the game is only a minor goal.

4. Comments

My main.py to play connect 4 is copied almost line-by-line from the provided C++ program, with a few changes described below. I also made some changes to the suggested interface as I was running into many errors with inconsistencies between the main board and the player's board.

- Instead of maintaining its own state of the board and updating it after each turn, the RandPlayer and AgentPlayer take the current state of the board as parameters (board and top). This ensures the state of the two boards is the same, and reduces the need for updating a board three times. I made this change because I had some minor issues with the game board not being in sync between the players and the main. Instead of spending time trying to debug that I just implemented this solution.
- The Agent Player takes a marble, a name modifier and a depth as arguments when it is created.
- I added a human player for testing. It can be instantiated the same way as RandPlayer. It is found in HumanPlayer.py
- Many helper functions were moved to Board.py
- The constants that define the game board were moved to Board.py. They can be edited there.

Main creates an AgentPlayer and calls agentPlayer.move, and agent player calls adversarialSearch to find the optimal move.