CSE 5360 AI-1
Assignment- 4
Evaluation function explanation

Name: Sagar Sharma
UTA ID: 1001626958

The eval function in the minimax program calculated the utility value for a non-terminal state based on the scores for each player in that sub board state which in turn is based up on the rules of the maxconnect4 game for sub board states which are bounded by the depth limit provided in the arguments of the command to run the program.

Also, these approx. utility values for those sub-board states measured by the evaluation function are pushed up to the corresponding min and max states and then the maximizing as well as minimizing operations are performed based on the min/max player state using the minVal() and maxVal() methods. Also, the alpha-beta strategy is used to truncate the sub-board states which are of no use in deciding the max value which can be achieved when the opponent plays the best possible move to minimize the max players score which in this scenario is the computer-player and the opponent is the human player. This decision is taken by the makedecision() method in the minimax class.

Note: unoccupied spaces are considered in the weighted sum if it is in the same slope as the sequential moves.

The evaluation function used for the depth limited minimax works on the weighted sum of the sequential moves(horizontal, vertical or diagonal) which works on the combination of highest weight for the 4 sequential moves, 2nd highest weight for the 3 sequential moves and 1 unoccupied space in the same slope as the sequential moves, 3rd highest weight for the 2 sequential moves and 2 unoccupied spaces, least highest weight for 1 move and 3 sequential unoccupied spaces for either of the two players and the utility value for that board state is calculated based on if max player/computer player has the higher score than the human player or vice versa.

Code that implements the logic for the eval function is provided below:

```
def result(oldGame, column):
    newGame = maxConnect4Game()

    try:
        newGame.nodeDepth = oldGame.nodeDepth + 1
    except AttributeError:
        newGame.nodeDepth = 1

    newGame.pieceCount = oldGame.pieceCount
    newGame.gameBoard = copy.deepcopy(oldGame.gameBoard)
    if not newGame.gameBoard[0][column]:
        for i in range(5, -1, -1):
            if not newGame.gameBoard[i][column]:
                newGame.gameBoard[i][column] = oldGame.currentTurn
                newGame.pieceCount += 1
                break
    if oldGame.currentTurn == 1:
        newGame.currentTurn = 2
    elif oldGame.currentTurn == 2:
        newGame.currentTurn = 1

    newGame.checkPieceCount()
    newGame.countScore()

    return newGame

##implementing alpha-beta pruned depth limited minimax algorithm
class Minimax:
    def __init__(self, game, depth):
        self.currentTurn = game.currentTurn
        self.game = game
        self.maxDepth = int(depth)
## alpha beta pruning with depth limited implemented
    def makeDecision(self): ## returns an action leading to the next board state with a utility v

        minValues = [] ## define as a empty list
        possMoves = possibleMoves(self.game.gameBoard)  ## possible moves for the board
state

        for move in possMoves:

            rslt = result(self.game,move)
            minValues.append( self.minVal(rslt,99999,-99999) ) ##minval function called to
figure out the next best action

        chosen = possMoves[minValues.index( max( minValues ) )] ## action chosen
        return chosen
## minval helps to get the max utility value for the best opponent moves
    def minVal(self, state, alpha, beta):
        if state.pieceCount == 42 or state.nodeDepth == self.maxDepth: ## check for the
terminal state
            return self.utility(state) ## return the utility value
        v = 99999   ## max value possible for alpha
        for move in possibleMoves(state.gameBoard):  ## iterate through the possible moves
            newState = result(state,move)  ## new states iterate through the possible
moves

            v = min(v,self.maxVal( newState,alpha,beta ))
            if v <= alpha: ## trunkate the next possible sub board state if true
                return v
            beta = min(beta, v) ## if not update the beta value
        return v
## maxval returns the best utility value possible for the best opponent moves
    def maxVal(self, state, alpha, beta):
        if state.pieceCount == 42 or state.nodeDepth == self.maxDepth: ## check fro the
terminal state
            return self.utility(state) ## return the utility value
        v = -99999 ## min value for beta

        for move in possibleMoves(state.gameBoard):  ## iterate through the possible moves
            newState = result(state,move)   ## new states iterate through the possible
moves

            v = max(v,self.minVal( newState,alpha,beta ))
            if v >= beta: ##trunkate the next possible sub board state if true
                return v
            alpha = max(alpha, v) ## if not update alpha value
        return v

    def utility(self,state): ## utility value function
        if self.currentTurn == 1: ## if player 1
            utility = state.player1Score * 2 - state.player2Score
        elif self.currentTurn == 2: ## if player 2
            utility = state.player2Score * 2 - state.player1Score

        return utility
```