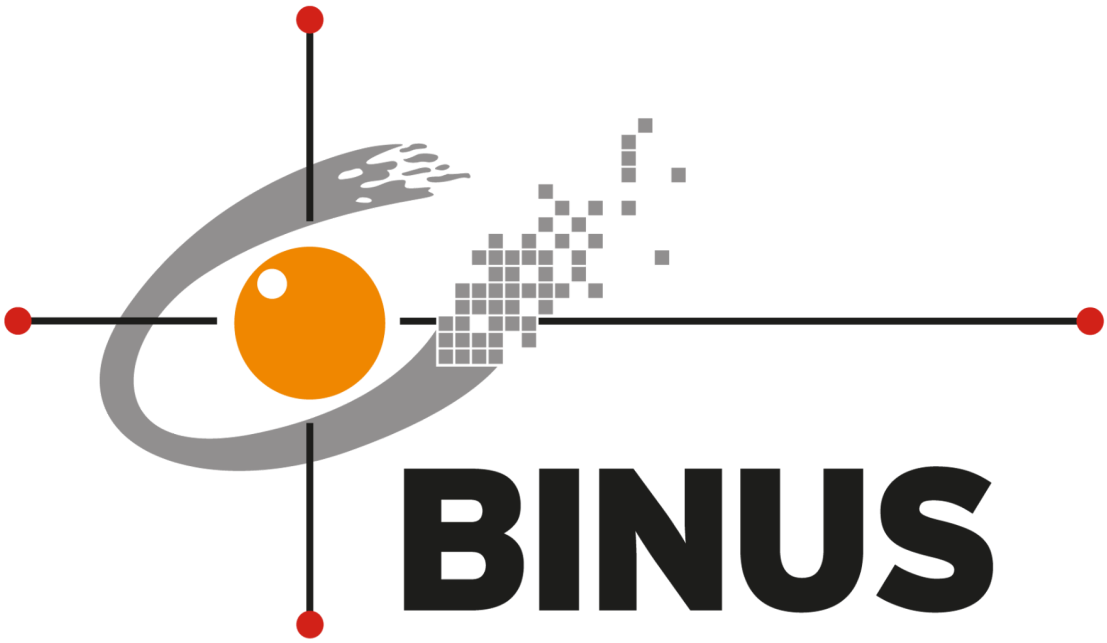


FINAL PROJECT ALGORITHM PROGRAMMING

Lecturer: JUDE JOSEPH LAMUG MARTINEZ , MCS

Name: I Kadek Gita Pradnya Widagda

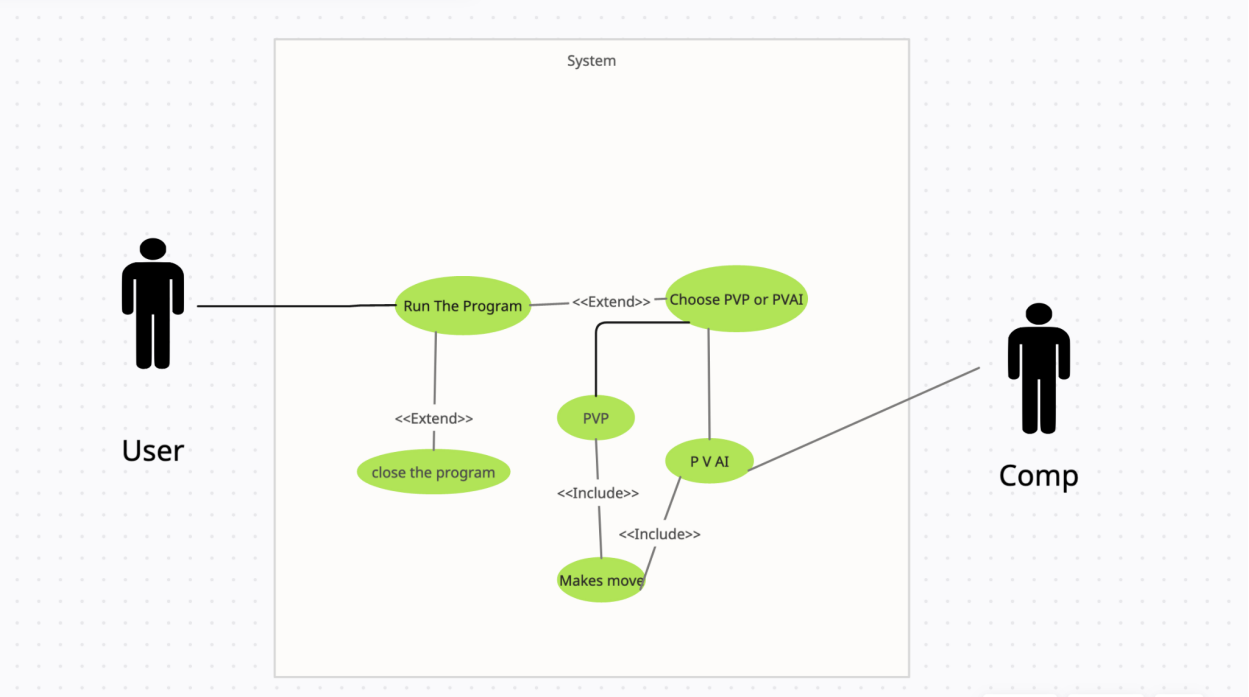
Student ID: 260211850

A. Project Documentation

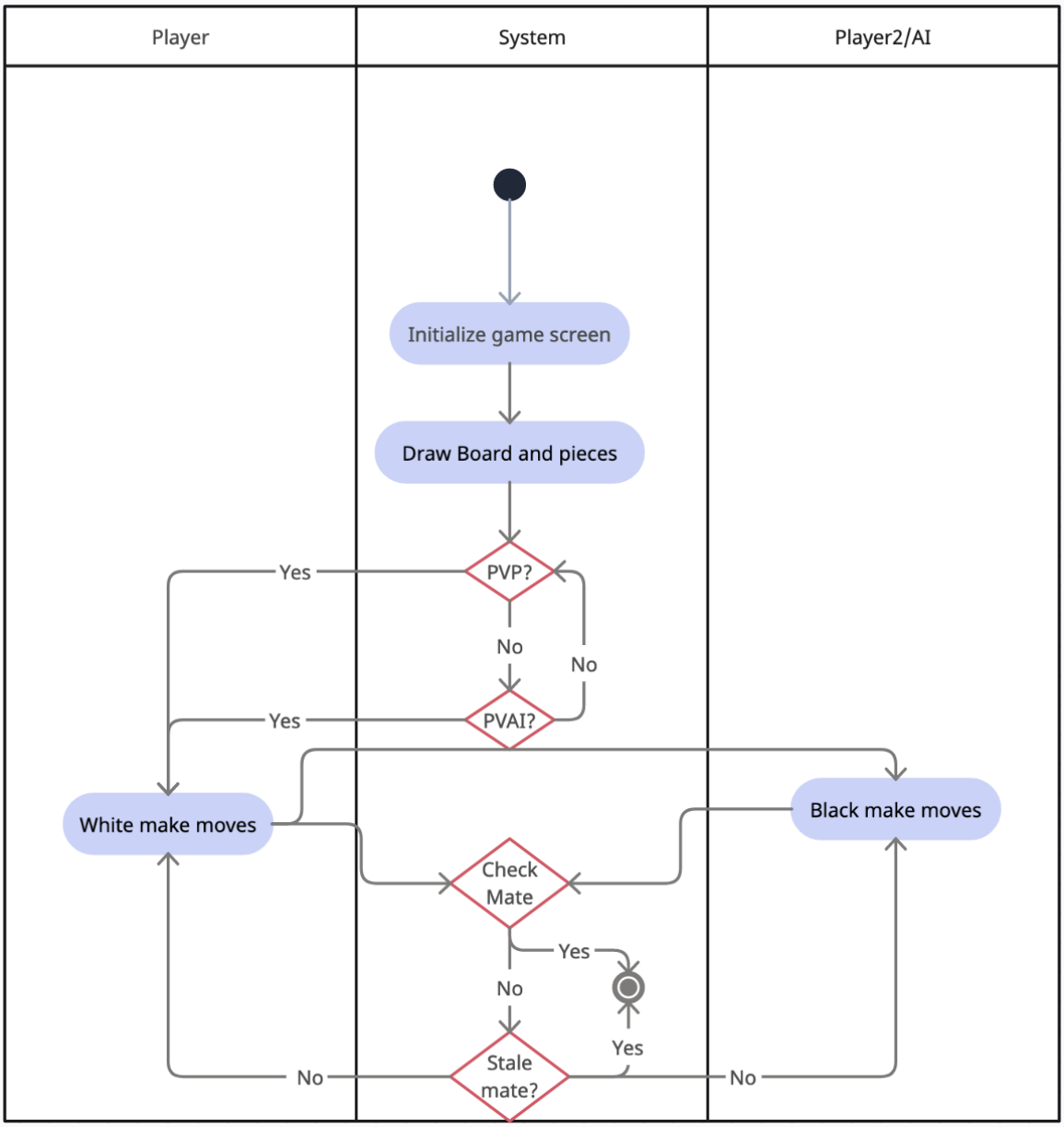
1. Brief Description

Chess is a board game of two people where the purpose of the game is to check the King. Usually this game is played by two people. In this project I am trying to make chess AI using the NegaMaxAlphaBetaPruning Algorithm.

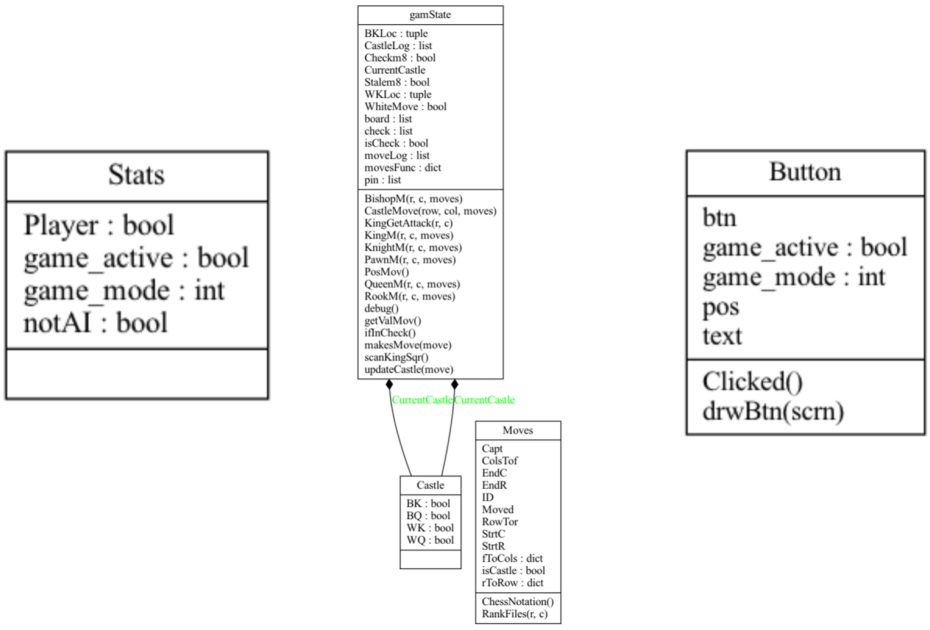
2. Use-case Diagram



3. Activity Diagram



4. Class Diagram



5. Modules

The modules that i use for this project are Pygame, Sys and Random

6. Essential Algorithms

a. AI Algorith

The algorithm that I'm using to make the AI is the NegaMaxAlphaBetaPruning algorithm. So NegaMax is the simplification of the MiniMax search tree algorithm. Alpha Beta Pruning is basically pruning of useless branches in decision trees.

The eval = -NegaMax is calling the function going to call the method again for the opponent. So whatever the best score for them is the worst score for the player. For example if black moves their best score is negative(-) and it’s going to negate that

score, that's why the negative symbol is the power of the NegaMax algorithm

```python
def NegaMax(gs, VMoves, depth, alpha, beta, turnToPlay):
    global nextMove
    if depth == 0:
        return turnToPlay * scoreBoard(gs)

    MaxScore = -mate
    for move in VMoves:
        gs.makesMove(move)
        nextMoves = gs.getValMov()
        eval = -NegaMax(gs, nextMoves, depth-1, -alpha, -beta, -turnToPlay)
        if eval > MaxScore:
            MaxScore = eval
            if depth == DEPTH:
                nextMove = move
        gs.debug()

        if MaxScore > alpha:
            alpha = MaxScore
        if beta <= alpha:
            break
    return MaxScore
```

(function) scoreBoard(gs: Any) -> (float | int)

b. Creating the board

```python
def __init__(self):
    self.board = [
    ["BR","BN","BB","BQ","BK","BB","BN","BR"],
    ["BP","BP","BP","BP","BP","BP","BP","BP"],
    [".."," ..",".."," ..",".."," ..",".."," .."],
    [" ..",".."," ..",".."," ..",".."," ..",".."],
    [".."," ..",".."," ..",".."," ..",".."," .."],
    [" ..",".."," ..",".."," ..",".."," ..",".."],
    ["WP","WP","WP","WP","WP","WP","WP","WP"],
    ["WR","WN","WB","WQ","WK","WB","WN","WR"]
    ]
```

```python
#load the image
def loadImg():
    pieces = ['WP','WN','WR','WB','WK','WQ','BP','BN','BR','BB','BK','BQ']
    for i in pieces:
        IMAGES[i] = p.transform.scale(p.image.load("images/"+ i + ".png"), (sqrSize, sqrSize))

#draw the board and pieces
def BoardPieces(scrn, gs, valMov, sqSlctd):
    drwBP(scrn,gs)
    MoveHighlight(scrn, gs, valMov, sqSlctd)

def drwBP(scrn, gs):
    colors = [p.Color("#FFFAF0"), p.Color("#333333")]
    for r in range(d):
        for c in range(d):
            color = colors[((r+c)%2)]
            p.draw.rect(scrn, color, p.Rect(c*sqrSize,r*sqrSize,sqrSize,sqrSize))
            piece = gs.board[r][c]
            if piece != "..":
                scrn.blit(IMAGES[piece],p.Rect(c*sqrSize,r*sqrSize,sqrSize,sqrSize))
```

c. Making the move and getting the valid move

```
    #to make the move
    def makesMove(self, move):
        self.board[move.StrtR][move.StrtC] = '..'
        self.board[move.EndR][move.EndC] = move.Moved
        self.moveLog.append(move)
        self.WhiteMove = not self.WhiteMove
        if move.Moved == 'WK':
            self.WKLoc = (move.EndR, move.EndC)
        elif move.Moved == "BK":
            self.BKLoc = (move.EndR, move.EndC)
        if move.isCastle:
            if move.EndC - move.StrtC == 2:
                self.board[move.EndR][move.EndC-1] = self.board[move.EndR][move.EndC+1]#this line and next line is to move the rook and
                self.board[move.EndR][move.EndC+1] = ".."
            else:
                    self.board[move.EndR][move.EndC+1] = self.board[move.EndR][move.EndC-2]
                    self.board[move.EndR][move.EndC-2] = ".."
        self.updateCastle(move)
        self.CastleLog.append(Castle(self.CurrentCastle.WK, self.CurrentCastle.BK, self.CurrentCastle.WQ, self.CurrentCastle.BQ))
```

```
class Moves():
    def __init__(self, StrtSq, EndSq, board, isCastle = False):
        self.StrtR = StrtSq[0]
        self.StrtC = StrtSq[1]
        self.EndR = EndSq[0]
        self.EndC = EndSq[1]
        self.Moved = board[self.StrtR][self.StrtC]
        self.Capt = board[self.EndR][self.EndC]
        self.ID = self.StrtR*10000+self.StrtC*1000+self.EndR*100+self.EndC
        self.isCastle = isCastle
    #overriding
    def __eq__(self, other):
        if isinstance(other, Moves):
            return self.ID == other.ID
        return False
```

```
    def getValMov(self):
        moves = []
        tmpCastle = Castle(self.CurrentCastle.WK, self.CurrentCastle.BK,self.CurrentCastle.WQ,self.CurrentCastle.BQ)
        self.isCheck, self.pin, self.check = self.scanKingSqr()
        if self.WhiteMove:
            KR = self.WKLoc[0]
            KC = self.WKLoc[1]
        else:
            KR = self.BKLoc[0]
            KC = self.BKLoc[1]
        if self.isCheck:
            if len(self.check) == 1:
                moves = self.PosMov()
                checks = self.check[0]
                CRow = checks[0]
                CCol = checks[1]
                pieceThatChecks = self.board[CRow][CCol]
                valSqr = []
                if pieceThatChecks[1] == "N":
                    valSqr = [(CRow,CCol)]
                else:
                    for i in range(1,8):
                        valSqrs = (KR + checks[2]*i, KC + checks[3]*i)
                        valSqr.append(valSqrs)
                        if valSqrs[0] == CRow and valSqrs[1] == CCol:
                            break
                for i in range(len(moves) -1,-1,-1):
                    if moves[i].Moved[1] != "K":
                        if not(moves[i].EndR, moves[i].EndC) in valSqr:
                            moves.remove(moves[i])
            else:
                self.KingM(KR,KC,moves)
        else:
            moves = self.PosMov()
            if self.WhiteMove:
                self.CastleMove(self.WKLoc[0], self.WKLoc[1], moves)
            else:
                self.CastleMove(self.BKLoc[0], self.BKLoc[1], moves)
        if len(moves) == 0:
            if self.ifInCheck():
                self.Checkm8 = True
            else:
                self.Stalem8 = True
        self.CurrentCastle = tmpCastle
        return moves
```

d. Castling move

```
def CastleMove(self, row, col, moves):
    if self.KingGetAttack(row, col):
        return
    if (self.WhiteMove and self.CurrentCastle.WK) or (not self.WhiteMove and self.CurrentCastle.BK):
        if self.board[row][col+1] == '..' and self.board[row][col+2] == '..':
            if not self.KingGetAttack(row, col+1) and not self.KingGetAttack(row, col+2):
                moves.append(Moves((row,col), (row,col+2), self.board, isCastle=True))
    if (self.WhiteMove and self.CurrentCastle.WQ) or (not self.WhiteMove and self.CurrentCastle.BQ):
        if self.board[row][col-1] == '..' and self.board[row][col-2] == '..' and self.board[row][col-3] == '..':
            if not self.KingGetAttack(row, col-1) and not self.KingGetAttack(row, col-2):
                moves.append(Moves((row,col), (row,col-2), self.board, isCastle=True))

def updateCastle(self, move):
    if move.Moved == "BK":
        self.CurrentCastle.BK = False
        self.CurrentCastle.BQ = False
    elif move.Moved == "WK":
        self.CurrentCastle.WK = False
        self.CurrentCastle.WQ = False
    elif move.Moved == "BR":
        if move.StrtR == 0:
            if move.StrtC == 0:
                self.CurrentCastle.BQ = False
            elif move.StrtC == 7:
                self.CurrentCastle.BK = False
    elif move.Moved == "WR":
        if move.StrtR == 7:
            if move.StrtC == 7:
                self.CurrentCastle.WK = False
            elif move.StrtC == 0:
                self.CurrentCastle.WQ = False
```

e. Creating move for every pieces

```
#setting the king moves
def  KingM(self,r,c,moves):
    dir = ((-1,0), (0,-1), (1,0), (0,1),(-1,-1), (-1,1), (1,-1), (1,1))
    ally = 'W' if self.WhiteMove else "B"
    for i in dir:
        for j in range(1,8):
            endRow = r + i[0]
            endCol = c + i[1]

            if 0<=endRow<8 and 0<=endCol<8:
                Pos = self.board[endRow][endCol]
                if Pos[0] != ally:
                    if ally == "W":
                        self.WKLoc = (endRow,endCol)
                    else:
                        self.BKLoc = (endRow,endCol)
                    isCheck, pin, check = self.scanKingSqr()
                    if not isCheck:
                        moves.append(Moves((r,c),(endRow,endCol),self.board))
                    if  ally == "W":
                        self.WKLoc = (r,c)
                    else:
                        self.BKLoc = (r,c)
```

```python
#setting the pawn moves
def PawnM(self,r,c,moves):
    isPinned = False
    pinDir = ()
    for i in range(len(self.pin)-1,-1,-1):
        if self.pin[i][0] == r and self.pin[i][1] == c:
            isPinned = True
            pinDir = (self.pin[i][2], self.pin[i][3])
            self.pin.remove(self.pin[i])
            break

    if self.WhiteMove:
        if self.board[r-1][c] == "..":
            if not isPinned or pinDir == (-1,0):
                moves.append(Moves((r,c),(r-1,c),self.board))
                if r == 6 and self.board[r-2][c] == "..":
                    moves.append(Moves((r,c),(r-2,c),self.board))
        if c-1 >= 0:
            if self.board[r-1][c-1][0] == 'B':
                if not isPinned or pinDir == (-1,-1):
                    moves.append(Moves((r,c),(r-1,c-1),self.board))
        if c+1 <= 7:
            if self.board[r-1][c+1][0] == 'B':
                if not isPinned or pinDir == (-1,1):
                    moves.append(Moves((r,c),(r-1,c+1),self.board))
    else:
        if self.board[r+1][c] == "..":
            if not isPinned or pinDir == (1,0):
                moves.append(Moves((r,c),(r+1,c),self.board))
                if r == 1 and self.board[r+2][c] == "..":
                    moves.append(Moves((r,c),(r+2,c),self.board))
        if c-1 >= 0:
            if self.board[r+1][c-1][0] == 'W':
                if not isPinned or pinDir == (1,-1):
                    moves.append(Moves((r,c),(r+1,c-1),self.board))
        if c+1 <= 7:
            if self.board[r+1][c+1][0] == 'W':
                if not isPinned or pinDir == (1,1):
                    moves.append(Moves((r,c),(r+1,c+1),self.board))
```

```python
#setting the rook moves
def RookM(self,r,c,moves):
    isPinned = False
    pinDir = ()
    for i in range(len(self.pin)-1,-1,-1):
        if self.pin[i][0] == r and self.pin[i][1] == c:
            isPinned = True
            pinDir = (self.pin[i][2], self.pin[i][3])
            if self.board[r][c][1] != "Q":
                self.pin.remove(self.pin[i])
            break
    dir = ((-1,0), (0,-1), (1,0), (0,1))
    enemy = 'B' if self.WhiteMove else "W"
    for i in dir:
        for j in range(1,8):
            endRow = r + i[0]*j
            endCol = c + i[1]*j

            if 0<=endRow<8 and 0<=endCol<8:
                if not isPinned or pinDir == i or pinDir == (-i[0], -i[1]):
                    Pos = self.board[endRow][endCol]
                    if Pos == '..':
                        moves.append(Moves((r,c),(endRow,endCol),self.board))
                    elif Pos[0] == enemy:
                        moves.append(Moves((r,c),(endRow,endCol),self.board))
                        break
                    else:
                        break
                else:
                    break
```

```
#setting the knight moves
def KnightM(self,r,c,moves):
    isPinned = False
    for i in range(len(self.pin)-1,-1,-1):
        if self.pin[i][0] == r and self.pin[i][1] == c:
            isPinned = True
            self.pin.remove(self.pin[i])
            break

    dir = ((-2,-1), (-2,1), (-1,-2), (-1,2), (1,-2),(1,2), (2,-1),(2,1))
    ally = 'W' if self.WhiteMove else "B"
    for i in dir:
        endRow = r + i[0]
        endCol = c + i[1]
        if 0<=endRow<8 and 0<=endCol<8:
            if not isPinned:
                Pos = self.board[endRow][endCol]
                if Pos == '..':
                    moves.append(Moves((r,c),(endRow,endCol),self.board))
                elif Pos[0] != ally:
                    moves.append(Moves((r,c),(endRow,endCol),self.board))
```

```
#setting the bishop moves
def  BishopM(self,r,c,moves):
    isPinned = False
    pinDir = ()
    for i in range(len(self.pin)-1,-1,-1):
        if self.pin[i][0] == r and self.pin[i][1] == c:
            isPinned = True
            pinDir = (self.pin[i][2], self.pin[i][3])
            self.pin.remove(self.pin[i])
            break
    dir = ((-1,-1), (-1,1), (1,-1), (1,1))
    enemy = 'B' if self.WhiteMove else "W"
    for i in dir:
        for j in range(1,8):
            endRow = r + i[0]*j
            endCol = c + i[1]*j

            if 0<=endRow<8 and 0<=endCol<8:
                if not isPinned or pinDir == i or pinDir == (-i[0], -i[1]):
                    Pos = self.board[endRow][endCol]
                    if Pos == '..':
                        moves.append(Moves((r,c),(endRow,endCol),self.board))
                    elif Pos[0] == enemy:
                        moves.append(Moves((r,c),(endRow,endCol),self.board))
                        break
                    else:
                        break
            else:
                break

#setting the queen moves
def  QueenM(self,r,c,moves):
    self. RookM(r,c,moves)
    self. BishopM(r,c,moves)
```

7. Screenshots

8. Lesson learned

There is a lot of new knowledge that I learnt from making this project.

First, making chess is not as simple as it looks. The mechanics of chess itself are hard to code and confusing. It took me quite a long time to code the mechanics of chess itself.

Second, I learnt a new algorithm. Although my chess ai is not smart enough, I'm still proud of making this ai and I'm looking forward to improving it and learning a new algorithm to improve the ai.

B. Source Code

https://github.com/TaigahG/ChessAI/tree/main/Chess

C. Video

https://drive.google.com/drive/folders/1V4hxnK6faES03CCugENNNB3pN0vfiLQB?usp=share_link

D. Reference

https://stackoverflow.com/questions/65750233/what-is-the-difference-between-minimax-and-negamax

https://medium.com/dscvitpune/lets-create-a-chess-ai-8542a12afef

https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/

https://www.youtube.com/watch?v=l-hh51ncgDI&t=256s
https://github.com/stratzilla/chess-engine

F. Github Repo

https://github.com/TaigahG/ChessAI/tree/main/Chess