به نام خدا

تمرین کامپیوتری دوم

هوش مصنوعی

نگار مرادی

شماره دانشجویی: 810198543

مقدمه:

در این تمرین می خواهیم با استفاده از الگوریتم minmax راه حل مناسبی برای بازی خود ارائه دهیم.

در این بخش باید توابع minmax, getValidMoves and getAllMoves را پیاده سازی کنیم.

پیاده سازی:

```
def getValidMoves(self, piece):

    moves = {}

    if piece.king:
        stopup = max(piece.row - 3, -1)   # tahe zamin baray vaghti ke dare mire bala ya do khune jolotar
        stopdown = min(piece.row + 3, ROWS)   # tahe zamin baray vaghti ke dare miad paein ya do khune jolotar
        moves.update(self._traverseRight(piece.row -1, stopup, -1, piece.color, piece.col + 1))  # right up
        moves.update(self._traverseLeft(piece.row - 1, stopup, -1, piece.color, piece.col - 1))  # left up
        moves.update(self._traverseRight(piece.row + 1, stopdown, 1, piece.color, piece.col + 1))  #righr down
        moves.update(self._traverseLeft(piece.row + 1, stopdown, 1, piece.color, piece.col - 1))  # left donw


    else:
        if piece.color == WHITE:
            stopdown = min(piece.row + 3, ROWS)
            moves.update(self._traverseRight(piece.row + 1, stopdown, 1, piece.color, piece.col + 1))  #righr down
            moves.update(self._traverseLeft(piece.row + 1, stopdown, 1, piece.color, piece.col - 1))  # left down

        if piece.color == RED:
            stopup = max(piece.row - 3, -1)
            moves.update(self._traverseRight(piece.row -1, stopup, -1, piece.color, piece.col + 1))  # right up
            moves.update(self._traverseLeft(piece.row - 1, stopup, -1, piece.color, piece.col - 1))  # left up

    return moves
```

در این تابع چک میکنیم که هر قطعه چه حرکات مجازی می تواند انجام دهد.

در ابتدا نوع مهره را چک میکنیم. اگر مهره کینگ بود برای آن 4 جهت مجاز را چک میکنیم.

آرگومان های تابع برای حرکت به بالا به این صورت هستند:

در هر بار مهره می تواند 1 حرکت به بالا رود پس :

step: -1

مهره می توانید تا وقتی بالا رود که یا به سر جدول برسد یا دوبار حرکت کند پس ماکسیمم مقدار 2 حرکت به سمت بالا یا سر جدول را میگیریم.

max(rows - 3, -1)

rows - 3 برای این است که چک کنیم با 2 حرکت از جدول خارج نشود.

حرکت به چپ و راست هم به این صورت است:

برای حرکت به سمت چپ از traverseLeft و برای حرکت به سمت راست از traverseRight استفاده می کنیم.

آرگومان های تابع برای حرکت به سمت پایین به این صورت هستند:

در هر بار مهره می تواند 1 حرکت به سمت پایین رود پس :

step: 1

مهره می توانید تا وقتی پایین رود که یا به ته جدول برسد یا دوبار حرکت کند پس مینیمم مقدار 2 حرکت به سمت پایین یا ته جدول را میگیریم.

min(rows + 3, Row)

rows + 3 برای این است که چک کنیم با 2 حرکت از جدول خارج نشود.

حرکت به چپ و راست هم به این صورت است:

برای حرکت به سمت چپ از traverseLeft و برای حرکت به سمت راست از traverseRight استفاده می کنیم.

حال اگر مهره کینگ نبود رنگ ان را چک می کنیم برای رنگ سفید حرکات مجاز حرکت به سمت چپ پایین و یا راست پایین است. (دوباره مثل حالت پایین رفتن مهره ی کینگ عمل می کنیم)

حرکات مجاز برای مهره ی قرمز چپ بالا و یا راست بالا است. (مانند حالت بالا رفتن مهره ی کینگ عمل می کنیم.)

تابع getAllMoves:

```
def getAllMoves(board, color, game):

    moves = []
    allpiece = board.getAllPieces(color)  #get pieces

    for eachPiece in allpiece:
        allValidMoves = board.getValidMoves(eachPiece)   #get valid moves
        for move, skip in allValidMoves.items():
            newBoard = UpdateBoard(game, board, eachPiece, skip, move)
            moves.append(newBoard)

    return moves
```

در این تابع برای همه ی مهره های یک رنگ تابع getValidMoves را صدا میزنیم تا حرکات مجاز همه ی مهره ها را به دست آوریم.

تابع UpdateBoard:

```
def UpdateBoard(game, board, piece, skip, moveVal):

    board.draw(game.win)
    pygame.display.update()
    nBoard = deepcopy(board)
    pRow, pCol = piece.row, piece.col
    nPiece = nBoard.getPiece(pRow, pCol)

    return simulateMove(nPiece, moveVal, nBoard, game, skip)
```

در این تابع برد را در ویندوز میکشیم و برد را اپدیت کرده و حرکت را برای مهره اجرا میکنیم. همچنین تابع simalateMove را صدا زده تا مختصات مهره را اپدیت کنیم.

تابع simulateMove:

```
def simulateMove(piece, move, board, game, skip):
    x, y = move
    board.move(piece, x, y)

    if skip:
        board.remove(skip)

    return board
```

در این تابع مختصات مهره را که حرکت کرده است اپدیت میکنیم و اگر از روی مهره ای پریده بود آن مهره را از برد پاک می کنیم.

تابع minmax:

```
def minimax(position, depth, maxPlayer, game):

    if maxPlayer:
        return maxi(position, game, depth)

    else:
        return mini(position, game, depth)
```

در این تابع الگوریتم سرچ را پیاده سازی می کنیم. در ابتدا چک میکنیم که کسی نبرده باشد و عمق ما 0 نباشد. اگر شرط ها برقرار نبود و نوبت مهره ی سفید بود وارد تابع maxi می شویم در غیر این صورت وارد تابع mini می شویم. اگر شرط ها برقرار بود evaluate برد و خود برد را برمیگردانیم.

تابع evaluate برای هر مهره در جدول یک وزن در نظر گرفته است که AI را تشویق کند که مهره هایش را به مهره کینگ تبدیل کند. و برای هر حرکت یک امتیاز به ما برمیگرداند. در واقع به این صورت عمل میکند که برای هر حالت مهره ها در جدول امتیازی بر میگرداند و AI از این امتیاز برای انتخاب حرکات بهتر استفاده می کند. (مثلا زدن مهره ی حریف یا تبدیل مهره به مهره ی کینگ). عبارت جمع در تابع برای مهره ی کینگ برای تشویق AI به تبدیل مهره هایش به مهره کینگ است.

تابع main

در تابع main یک قسمت اضافه کردیم که اگر حریفی مجاز به حرکت نبود جدول را تغییر ندهد.

```
        if game.turn == WHITE:
            value, newBoard = minimax(game.getBoard(), WHITE_DEPTH, True, game)
            if newBoard is None:
                newBoard = game.getBoard()
            game.aiMove(newBoard)

        if game.winner() != None:
            print(game.winner())
            run = False

        if game.turn == RED:
            value, newBoard = minimax(game.getBoard(), RED_DEPTH, False, game)
            if newBoard is None:
                newBoard = game.getBoard()
            game.aiMove(newBoard)
```

تابع mini

```
def mini(position, game, depth):

    if position.winner() != None or depth == 0 :
        return position.evaluate(), position

    bestMove, minim = None, inf
    moves = getAllMoves(position, RED, game)
    for move in moves:
        newDepth = depth - 1
        evalu, bMove = maxi(move, game, newDepth)
        if minim >= evalu and (evalu!= -inf or minim == inf):
            if minim >= evalu:
                bestMove = move
                minim = evalu

    return minim, bestMove
```

در این تابع ابتدا چک میکنیم که عمق ما 0 نباشد و کسی برنده نشده باشد. اگر شر ها برقرار نبود برای هر حرکت مجاز حرکت مهره ی حریف را بررسی میکنیم و یک evaluate برای ان در نظر میگیریم. اگر حرکتی باعث شود که مهره ی حریف evaluate کم تری داشته باشد آن حرکت، حرکت بهتری است.

متغیر bestmove را اپدیت کرده و مقدار evaluate را در مقدار minim که evaluate بهترین حرکت را نگه میدارد میریزیم.

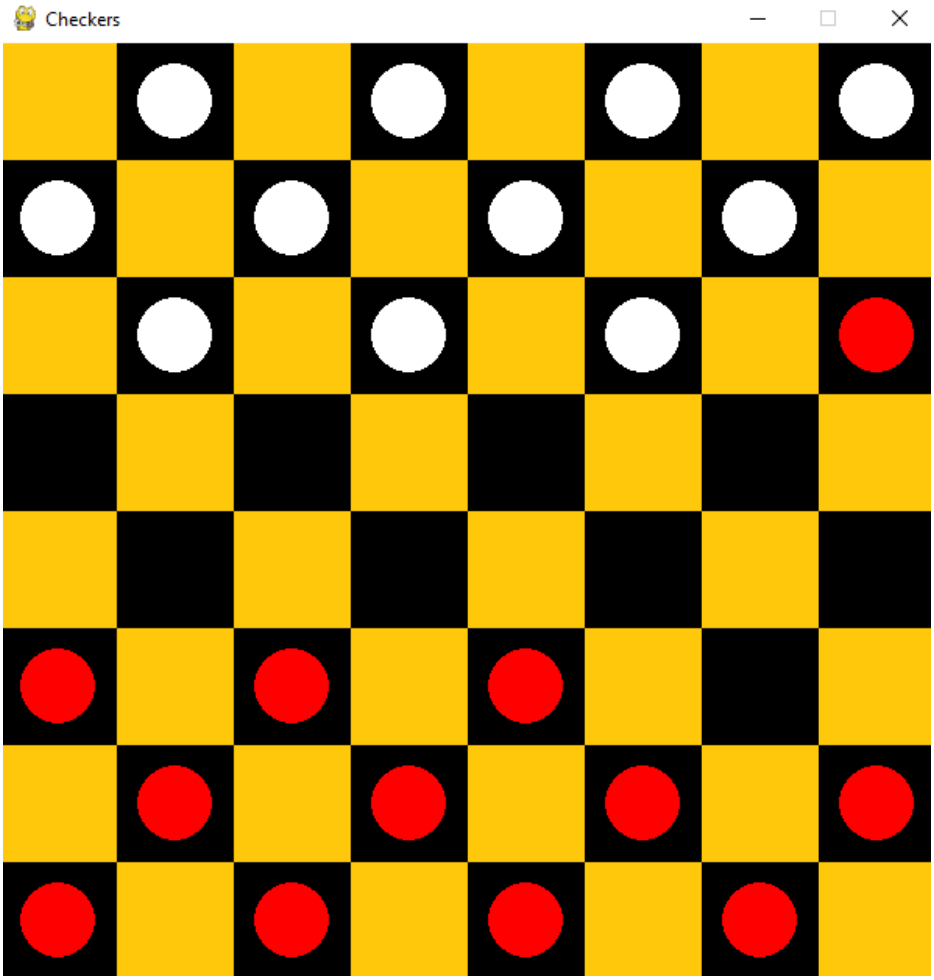
تابع maxi:

```
def maxi(position, game, depth):

    if position.winner() != None or depth == 0 :
        return position.evaluate(), position

    bestMove, maxim = None, -inf
    moves = getAllMoves(position, WHITE, game)
    for move in moves:
        newDepth = depth - 1
        evalu, bMove = mini(move, game, newDepth)
        if maxim <= evalu and (evalu != inf or maxim == -inf):
            if maxim <= evalu:
                bestMove = move
                maxim = evalu

    return maxim, bestMove
```

در این تابع هم مانند تابع mini ابتدا شرط پایان سرچ را چک میکنیم و سپس برای حرکات مجاز برای مهره سفید حرکات مهره قرمز را با عمق چک میکنیم و مقدار evaluate را مثل تابع قبل اپدیت میکنیم.

نتایج:

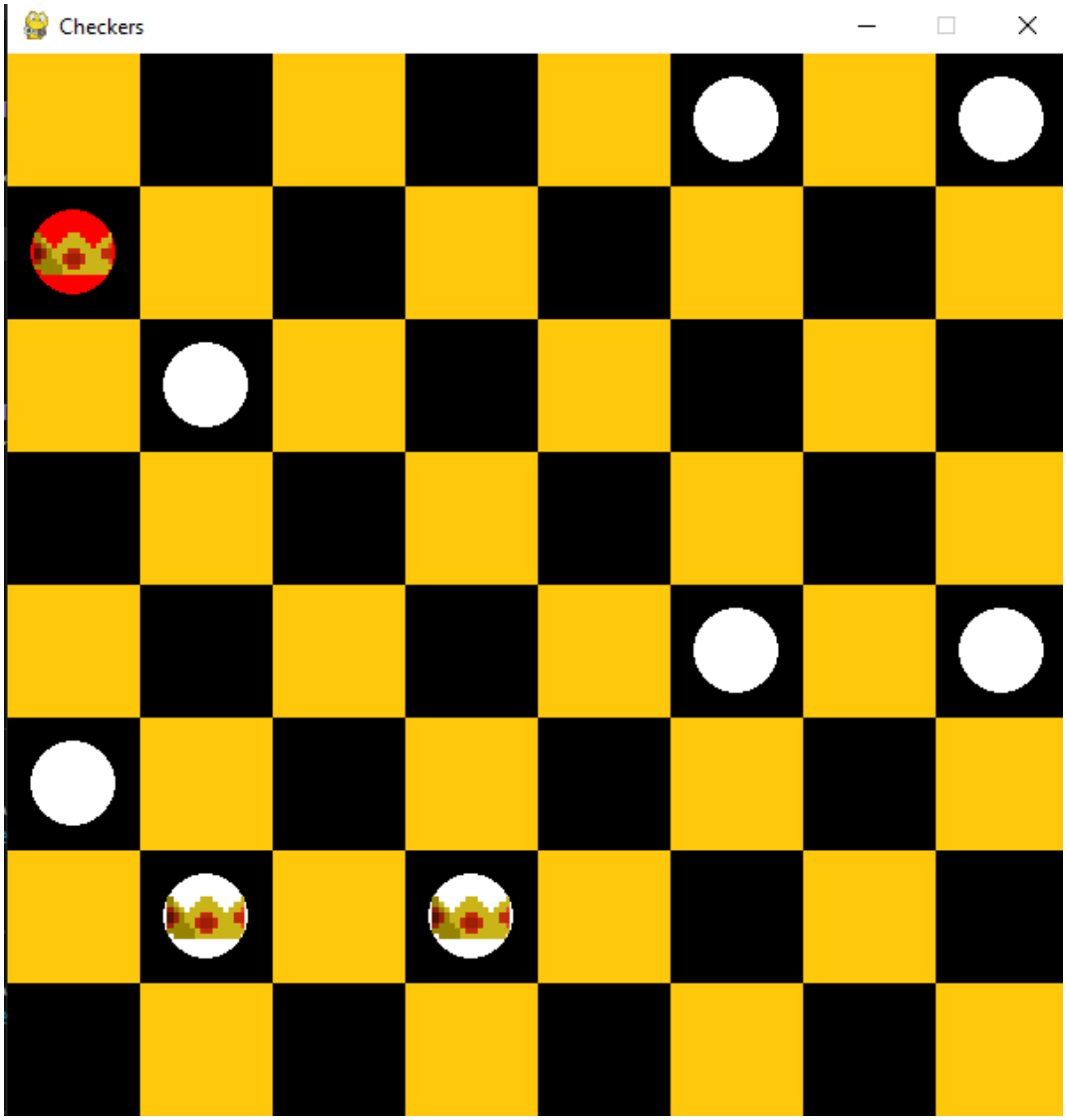
تست برای عمق white:1 , red:1



در این حالت مهره ها همگام باهم پیش می روند ولی در انتها مهره ی قرمز برنده می شود.

```
pygame 2.0.3 (SDL 2.0.16, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
(255, 0, 0)
```

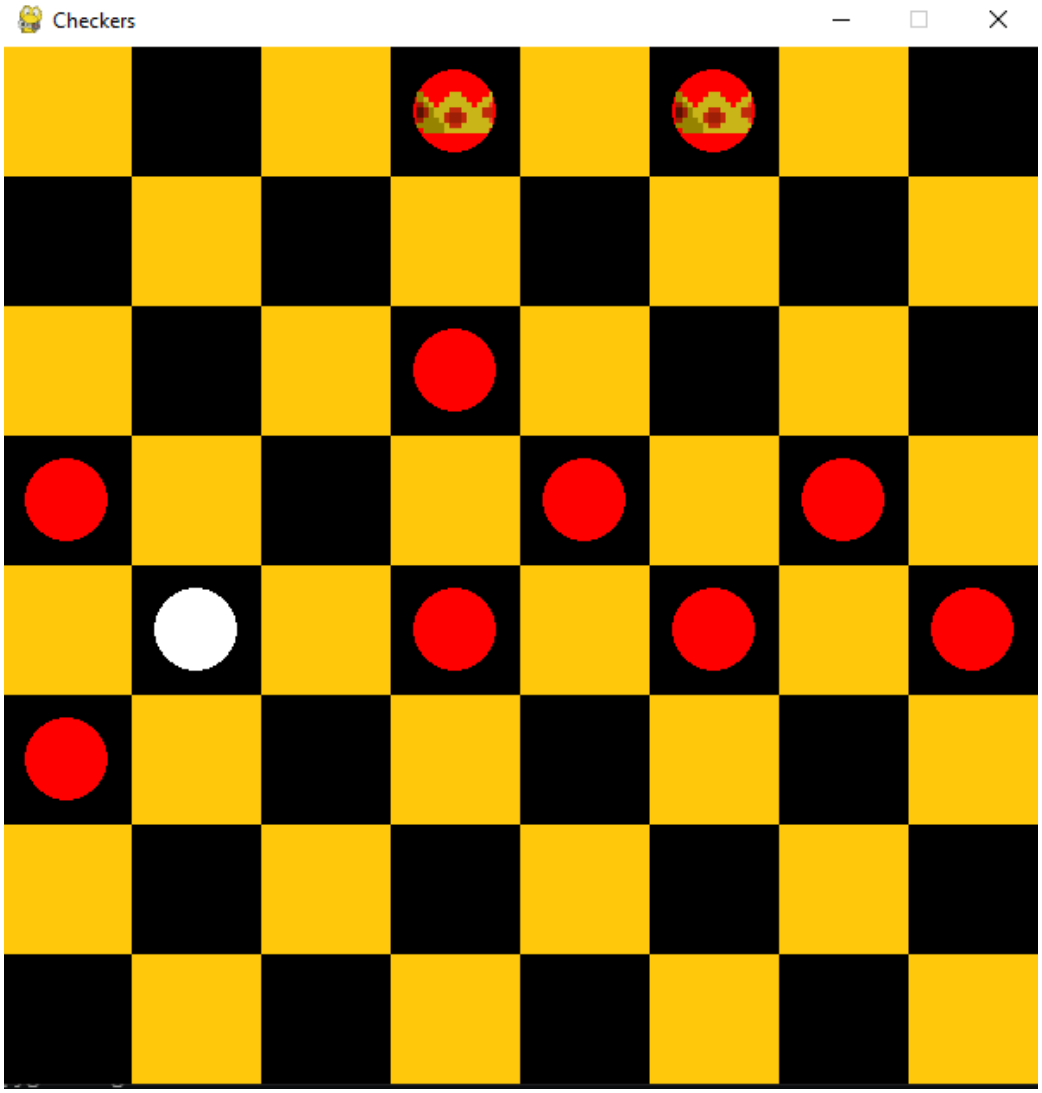
تست برای عمق white:3 and red:1



در این حالت مهره ی سفید برنده بازی می شود.

```
pygame 2.0.3 (SDL 2.0.16, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
(255, 255, 255)
```

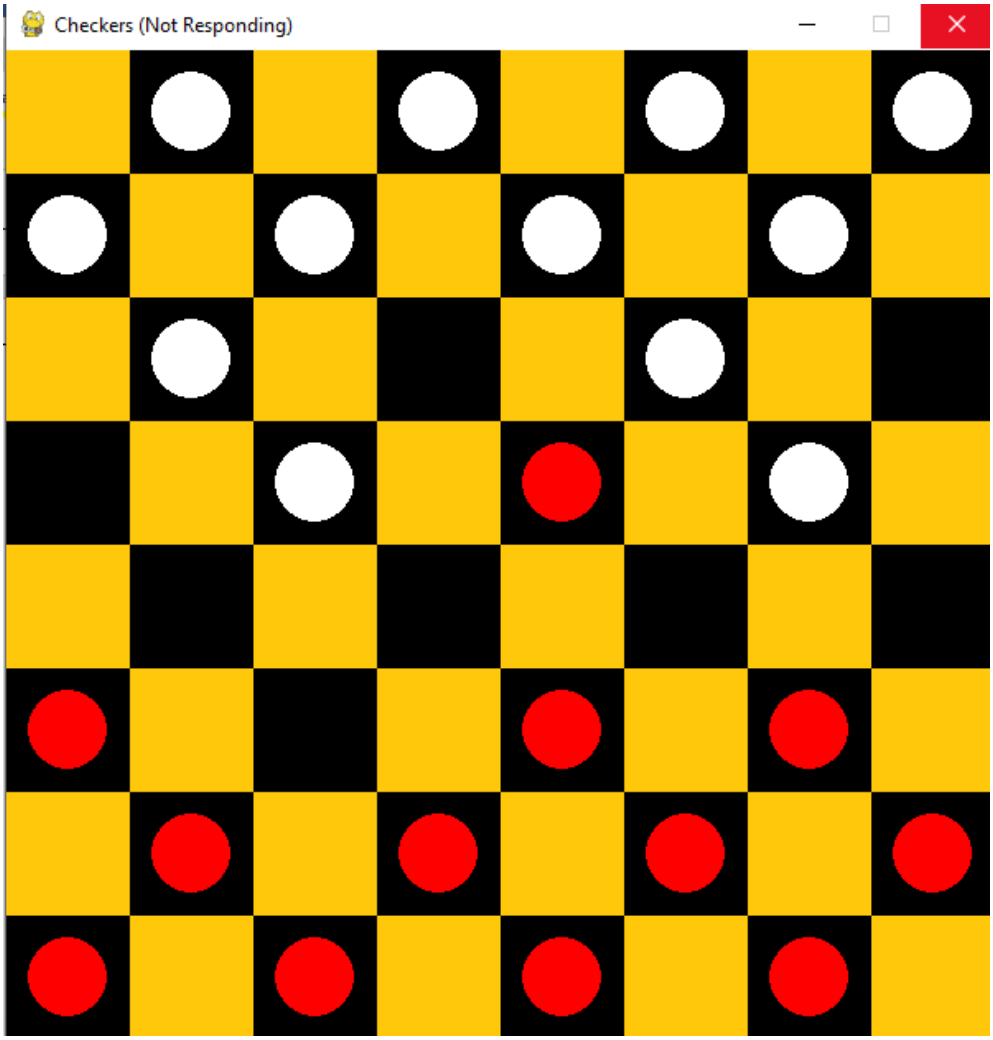
تست برای white:1 red:3




```
pygame 2.0.3 (SDL 2.0.16, Python 3.8.5)
Hello from the pygame community. https://www.pygame.org/contribute.html
(255, 0, 0)
PS C:\Users\negar\OneDrive\Documents\vs code\AI\CA2>
```

مهره ی قرمز می برد.

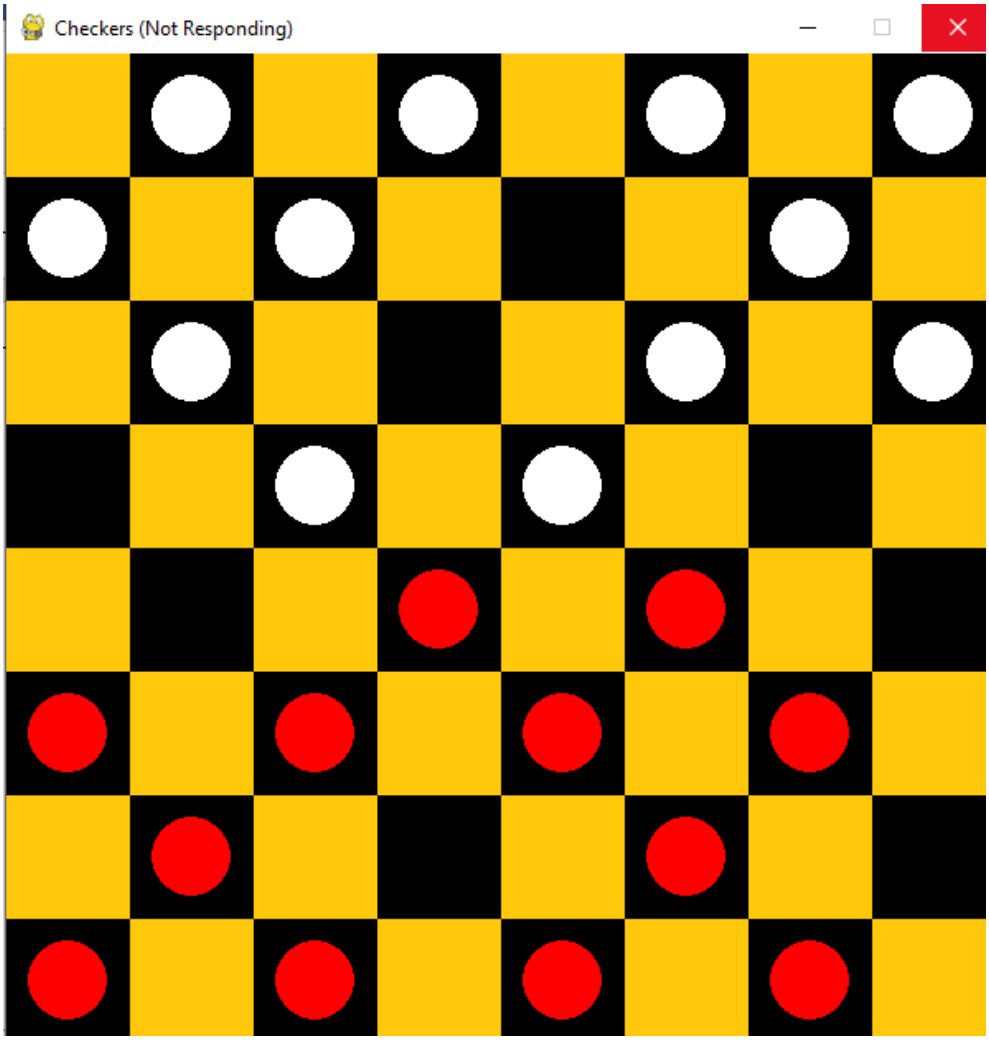
تست برای white:2 red:5



(تصویر مال وقتی که تو لوپ میفته نیست وقتی تو لوپ میفته تعداد سفیدا کم تره)

بازی تا جایی پیش ی رود و بعد از آن در لوپ می افتد.

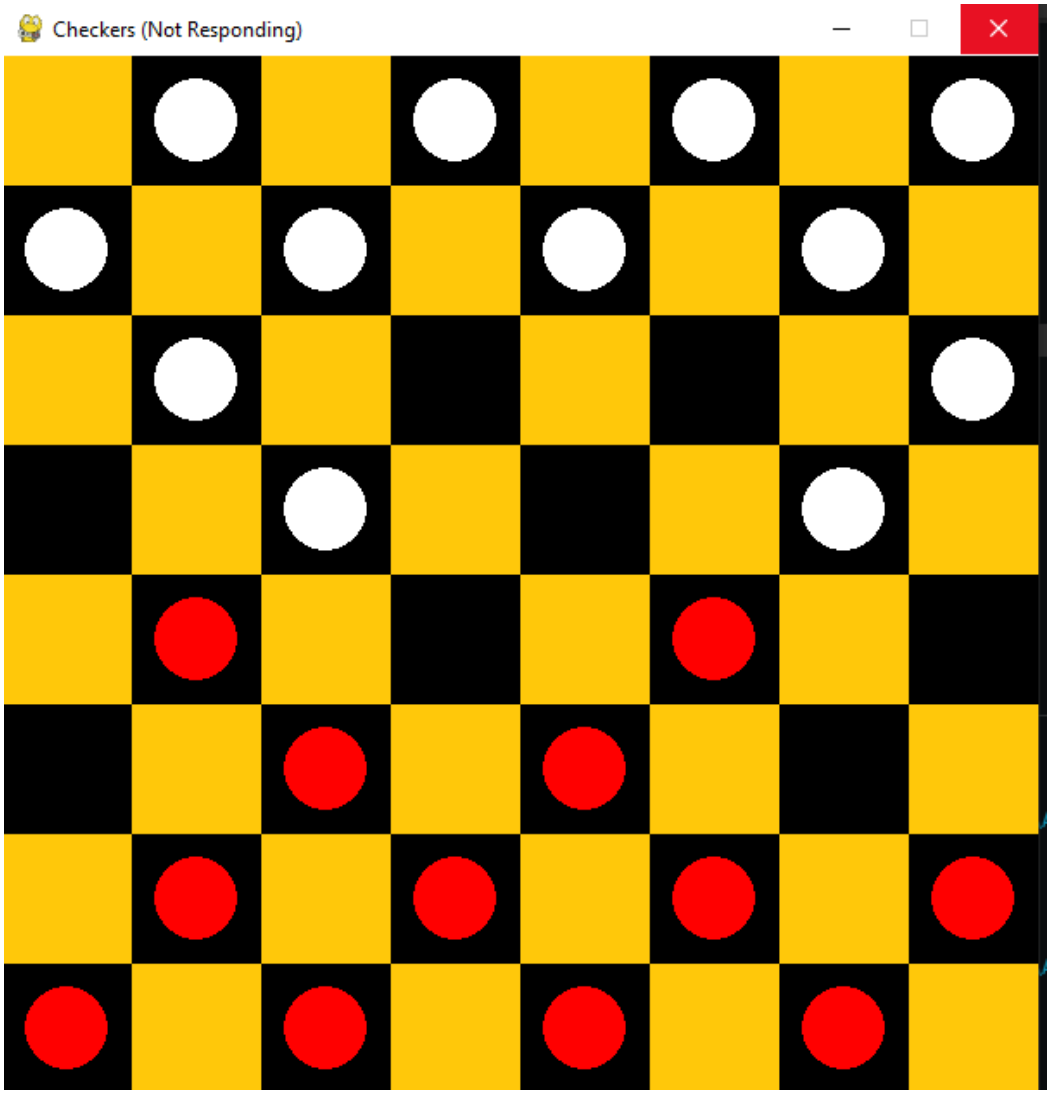
تست برای white:5 red:2



(تصویر مال وقتی که تو لوپ میفته نیست وقتی تو لوپ میفته تعداد سفیدا بیش تره)

بازی تا جایی پیش ی رود و بعد از آن در لوپ می افتد.

تست برای white:5 red:5



بازی تا جایی پیش می رود و بعد از آن در لوپ می افتد.

تحلیل:

در مثال دوم و سوم و چهارم و پنجم عمق برای مهره های رنگ قرمز و رنگ سفید یکسان نیست. طبق مثال دوم و سوم می توان گفت که هرچقدر عمق سرچ برای یک مهره بیشتر باشد برای حرکت بعدی آن مهره اشتباه کم تری صورت میگیرد و تصمیم بهتری میگیرد زیرا می تواند حرکت های بیشتری از مهره ی حریف را پیش بینی کند. در نتیجه همانطور که در مثال میبینیم وقتی عمق قرمز بیشتر است قرمز میبرد و وقتی عمق سفید بیشتر است سفید میبرد. در مثال چهارم و پنجم به دلیل انجام حرکت تکراری بازی در لوپ می افتد در نتیجه قادر به مشاهده ی نتیجه ی آن نیستیم.

زمانی که عمق هر دو مهره های قرمز و سفید یکی باشد انتظار داریم که مهره ای که زودتر بازی را شروع کرده ببرد. اگر برنامه یک سری قوانین و حرکات مشخص داشته باشد برتری با مهره ای است که اول بازی را شروع میکند.(در مثال 1 مهره ی قرمز میبرد) در بقیه مواقع بازی تو لوپ می افتد چون یک سری حرکت دوباره تکرار میشوند. در عمق یکسان هر دو مهره تقریبا همگام پیش می روند.