شرح توابع :

۱.

```
def drop_piece(board, row, col, piece):
    board[row][col] = piece
```

این تابع اگر شرایط مهره گذاری (که به آن ها پرداخته خواهد شد) را داشته باشد صفحه بازی را با مهره از نوع piece پر خواهد کرد

۲.

```
def is_valid_location(board, col):
    return board[ROW_COUNT - 1][col] == 0
```

این تابع بررسی خواهد کرد که آیا می توان در صفحه بازی و ستون col مهره ای قرار داد یا خیر

۳.

```
def get_next_open_row(board, col):
    for r in range(ROW_COUNT):
        if board[r][col] == 0:
            return r
```

این تابع با ایجاد حلقه ای جستجو میکند که در ستون col در چه سطری جای خالی وجود دارد تا آن را با مهره بازی پر کند

۴.

```
def winning_move(board, piece):
    # Check horizontal locations for win
    for c in range(COLUMN_COUNT-3):
        for r in range(ROW_COUNT):
            if board[r][c] == piece and board[r][c + 1] == piece and board[r][c + 2] == piece and board[r][c + 3] == piece:
                return True

    # Check vertical locations for win
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT-3):
            if board[r][c] == piece and board[r + 1][c] == piece and board[r + 2][c] == piece and board[r + 3][c] == piece:
                return True

    # Check positively sloped diaganols
    for c in range(COLUMN_COUNT-3):
        for r in range(ROW_COUNT-3):
            if board[r][c] == piece and board[r + 1][c + 1] == piece and board[r + 2][c + 2] == piece and board[r + 3][c + 3] == piece:
                return True

    # Check negatively sloped diaganols
    for c in range(COLUMN_COUNT-3):
        for r in range(3, ROW_COUNT):
            if board[r][c] == piece and board[r - 1][c + 1] == piece and board[r - 2][c + 2] == piece and board[r - 3][c + 3] == piece:
                return True
```

در این تابع ما بررسی میکنیم که بازی به پایان رسیده است یا خیر. بدین صورت که تمام حالاتی که امکان دارد یک ردیف حداقل چهار تایی از مهره piece در بازی وجود دارد یا خیر. به صورت چهار حلقه تودرتو نوشته شده است که اولی افقی، دومی عمودی، سومی اُریب با شیب مثبت و چهارمی اُریب با شیب منفی را بررسی خواهد کرد. هر کدام که در صفحه موجود باشند این تابع مقدار True را بر میگرداند

پ.ن : در پیاده سازی این بازی علاوه بر بررسی کردن ردیف های افقی و عمودی، ردیف های اُریب هم بررسی شده است(نمره امتیازی)

۵.

```
def evaluate_window(window, piece):
    heuristic = 0
    opp_piece = PLAYER_PIECE
    if piece == PLAYER_PIECE:
        opp_piece = AI_PIECE

    if window.count(piece) == 4:
        heuristic += 100
    elif window.count(piece) == 3 and window.count(EMPTY) == 1:
        heuristic += 5
    elif window.count(piece) == 2 and window.count(EMPTY) == 2:
        heuristic += 2

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        heuristic -= 4

    return heuristic
```

در ابتدا این تابع یک لیست به نام window و یک مهره به نام piece میگیرد و در ابتدای کار مشخص میکند که این مهره، مهره بازیکن است یا مهره agent در گام بعدی به شرایط پیش می آید. اول این ک اگر ردیف ۴ تایی(four in a row) از مهره ها داشتیم به مقدار heuristic ۱۰۰ واحد اضافه می شود. دوم اگر سه مهره در یک ردیف داشتیم (three in a row) مقدار heuristic با ۱۰ جمع می شود و در نهایت اگر دو مهره پشت هم داشتیم (two in a row) مقدار heuristic با ۲ جمع می شود. اگر مهره حریف مقابل (بازیکن) حالت سه مهره پشت هم داشت یعنی این شاخه نباید بررسی شود زیرا در نهایت این شاخه منجر به شکست agent می شود پس باعث کاهش مقدار heuristic می شود.

توضیحات : این تابع در نهایت مقدار heuristic هر شاخه را محاسبه می کند با این تفاوت که این مقدار برعکس مرتب شده است. یعنی طوری پیاده سازی شده که شاخه با مقدار heuristic بیشتر بسط داده می شود. در روند کار تغییری ایجاد نمی شود زیرا نحوه heuristic دادن به هر شاخه معکوس است. مزیت آن این است که قابل فهم تر است و هر چقدر این میزان بیشتر باشد انگار بیشتر به برد agent نزدیک هستیم. (برای راحتی فرض شود مقدار مطلق heuristic استفاده شده است به عبارتی اگر heuristic آن بیشتر باشد انگار مقدار منفی بزرگ تری است و در نتیجه مقدار آن کمتر است و باید زودتر بسط داده شود)

۶.

```python
def heuristic_position(board, piece):
    heuristic = 0

    ## heuristic center column
    center_array = [int(i) for i in list(board[:, COLUMN_COUNT // 2])]
    center_count = center_array.count(piece)
    heuristic += center_count * 3

    ## heuristic Horizontal
    for r in range(ROW_COUNT):
        row_array = [int(i) for i in list(board[r, :])]
        for c in range(COLUMN_COUNT - 3):
            window = row_array[c: c + WINDOW_LENGTH]
            heuristic += evaluate_window(window, piece)

    ## heuristic Vertical
    for c in range(COLUMN_COUNT):
        col_array = [int(i) for i in list(board[:, c])]
        for r in range(ROW_COUNT - 3):
            window = col_array[r:r + WINDOW_LENGTH]
            heuristic += evaluate_window(window, piece)

    ## heuristic positive sloped diagonal
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board[r + i][c + i] for i in range(WINDOW_LENGTH)]
            heuristic += evaluate_window(window, piece)

    ## heuristic positive sloped diagonal
    for r in range(ROW_COUNT - 3):
        for c in range(COLUMN_COUNT - 3):
            window = [board[r + 3 - i][c + i] for i in range(WINDOW_LENGTH)]
            heuristic += evaluate_window(window, piece)

    return heuristic
```

در این تابع یک مقدار heuristic در ابتدا با صفر مقدار دهی شده است حال به چهار حلقه تودرتو نیاز داریم در هر کدام از حلقه ها نحوه پیدا کردن ردیف پشت هم(افقی، عمودی، اُریب با شیب مثبت، اُریب با شیب منفی) جستجو می شوند. در هر حلقه یک لیست چهارتایی به mode های مختلف به تابع evaluate_window فرستاده می شوند و این لیست های چهارتایی در کل صفحه iterate می شوند. در نهایت وقتی به صورت افقی به پایان رسیدیم از طرف عمودی مقدار heuristic را اپدیت میکنیم و در ادامه به نحوه اُریب مثبت و....

در ابتدا هم اگر مقدار heuristic صفر شد یک تابع پیشفرض از وسط صفحه را پر میکند (علت وسط بودن توضیح داده شود)

در نهایت این تابع مقدار heuristic شاخه را در تمام جهات بررسی کرده و به عنوان یک عدد به فانکشن بالاتر ارجاع میدهد.

۷.مینیمکس با کمک هرس آلفا-بتا

```python
def  minimaxab(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                return (None,  100000000000000)
            elif winning_move(board, PLAYER_PIECE):
                return (None, -100000000000000)
            else: # Game is over, no more valid moves
                return (None, 0)
        else: # Depth is zero
            return (None, heuristic_position(board, AI_PIECE))
    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
            new_heuristic = minimaxab(b_copy, depth-1, alpha, beta, False)[1]
            if new_heuristic > value:
                value = new_heuristic
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else: # Minimizing player
        value = math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, PLAYER_PIECE)
            new_heuristic = minimaxab(b_copy, depth - 1, alpha, beta, True)[1]
            if new_heuristic < value:
                value = new_heuristic
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value
```

در این تابع ابتدا تمام حرکت های مجاز را در یک لیست میریزیم (با استفاده از تابع get valid location که در ادامه توضیح داده خواهد شد) سپس با استفاده از سودوکد که در پایین آورده شده است، الگوریتم را تبدیل به کد پایتون میکنیم. بدین صورت که اگر به گره پایانی رسیده باشیم یا عمق شاخه بسط داده شده صفر شد، باید ببینیم که آیا agent بازی را پیروز شده است یا user هر کدام که بازی را برنده شده باشد، دو مقدار خروجی متفاوت است.(در کد مشخص است)

حال اگر به انتهای عمق نرسیده باشیم، باید بررسی کنیم که mode مینیمکس کردنمان باید بر اساس maximize باشد یا minimize، بدین منظور در هر کدام از حالات مقدار آن شاخه را در ابتدا برابر مثبت یا منفی بی نهایت قرار میدهیم و یک حرکت رندوم از بین حرکت های مجاز برای مقایسه بردارد سپس به ازای هر کدام از حرکات مجاز یک کپی از صفحه بازی بگیرد، آن حرکت را اعمال کند، مقدار heuristic آن را به دست بیاورد. سپس اگر state آن بهتر از استیت مقدار دهی شده اولیه باشد این مقدار را اپدیت کند، به عبارتی در نهایت در ابتدای حلقه مقدار بهینه ترین حالت را پیدا کرده و با استفاده از مقدار آلفا و بتا که در minimax کمک کرده اند، در نهایت مقدار بهینه ترین ستون و بهترین heuristic را خروجی میدهد

بخش minimize یا maximize کردن آن تقریبا مشابه هم است تنها تفاوت آن اعمال شرطی مقایسه ای آن است که یعنی باید مقدار min را پیدا کرد یا max

باقی جزئیات از روی سودوکد قابل تشخیص است

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            α := max(α, value)
            if value ≥ β then
                break (* β cutoff *)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            β := min(β, value)
            if value ≤ α then
                break (* α cutoff *)
        return value
```

۷/۱.

```python
def minimax(board, depth, maximizimg_player):
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 and is_terminal:
        if is_terminal:
            if winning_move(board, AI_PIECE):
                return (None,  1000000000000)
            elif winning_move(board, PLAYER_PIECE):
                return (None, -1000000000000)
            else: #game is over, no more valid moves
                return (None, 0)
        else: # depth is zero
            return (None, score_position(board, AI_PIECE))
    if maximizimg_player:
        value = -math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, AI_PIECE)
            new_score = minimax(b_copy, depth - 1, False)[1]
            if new_score > value:
                value = new_score
                column = col
            return column, value
    else: #Minimizing Player
        value = math.inf
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, PLAYER_PIECE)
            new_score = minimax(b_copy, depth - 1, True,)[1]
            if new_score < value:
                value = new_score
                column = col
            return column, value
```

این تابع بعد از تابع minmax هرس آلفا بتا پیاده سازی شده است. نحوه کار دقیقا مانند تابع قبلی می باشد با این تفاوت که هرس شاخه ای نداریم. یعنی در هر شاخه وقتی به طور بازگشتی فراخوانده می شود تمام شاخه های فرزند را بررسی کرده و بهترینِ آن را انتخاب میکند. بقیه پیاده سازی مانند تابع قبلی می باشد و فرقی با قبلی ندارد.

.۸

```
def is_terminal_node(board):
    return winning_move(board, PLAYER_PIECE) or winning_move(board, AI_PIECE) or len(get_valid_locations(board)) == 0
```

این تابع بررسی میکند که آیا در شاخه های درخت جستجو به گره پایانی رسیدیم یا خیر. برای فهم این موضوع باید سه موضوع را بررسی کنیم بدین منظور که اگر تابع winning_move مقدارTrue را بازگرداند و بازی تمام شود (دو حالت دارد) یا تمام صفحه پر شود و جایی برای مهره نداشته باشد(بازی مساوی تمام شود) آنگاه به گره پایانی رسیدیم و نمی توانیم بیش از این شاخه را بسط دهیم.

.۹

```
def get_valid_locations(board):
    valid_locations = []
    for col in range(COLUMN_COUNT):
        if is_valid_location(board, col):
            valid_locations.append(col)
    return valid_locations
```

در این تابع لیستی از تمام حرکات مجاز ساخته می شود که با استفاده از تابع is_valid_location بررسی خواهند شد و در نهایت این لیست به تابع های بالاتر ارجاع داده می شوند.

.۱۰

```
def draw_board(board):
    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            pygame.draw.rect(screen, BLUE, (c * SQUARESIZE, (r + 1) * SQUARESIZE, SQUARESIZE, SQUARESIZE))
            pygame.draw.circle(screen, WHITE, (int((c + 1/2) * SQUARESIZE), int((r + 3/2) * SQUARESIZE)), RADIUS)

    for c in range(COLUMN_COUNT):
        for r in range(ROW_COUNT):
            if board[r][c] == PLAYER_PIECE:
                pygame.draw.circle(screen, RED, (int((c + 1/2) * SQUARESIZE), height - int((r + 1/2) * SQUARESIZE)), RADIUS)
            elif board[r][c] == AI_PIECE:
                pygame.draw.circle(screen, YELLOW, (int((c + 1/2) * SQUARESIZE), height - int((r + 1/2) * SQUARESIZE)), RADIUS)
    pygame.display.update()
```

این تابع با استفاده از مقادیر صفحه بازی یک پنجره باز کرده در آن ابتدا به اندازه سایز مربع یا دایره هایی که در بازی موجود هستن، پنجره را با رنگ سفید پر میکند سپس برای پر کردن مقادیری که از قبل استفاده شده است با توجه به این که مال کدام یک از بازیکن یا agent هست، رنگ آن را مشخص میکند و صفحه را پر میکند، در نهایت صفحه را آپدیت کرده و نمایش می دهد

۱۱. بخش اصلی کد

اتبدا صفحه را با صفر پر کرده، pygame را تعریف میکنیم، و سایز پنجره را با توجه به مقادیر ثوابت مشخص میکنیم و در نهایت برنامه این قابلیت را دارد که شما به عتوان نفر اول بازی را شروع کنید یا به عنوالن نفر دوم بازی را ادامه دهید، و این به صورت تصادفی انجام می شود.

```
board = np.zeros((ROW_COUNT,COLUMN_COUNT))
game_over = False

pygame.init()

width = COLUMN_COUNT * SQUARESIZE
height = (ROW_COUNT + 1) * SQUARESIZE

size = (width, height)

RADIUS = int(SQUARESIZE / 2 - 5)

screen = pygame.display.set_mode(size)
draw_board(board)
pygame.display.update()

myfont = pygame.font.SysFont("monospace", 75)

turn = random.randint(PLAYER, AI)
```

بخش حلقه :

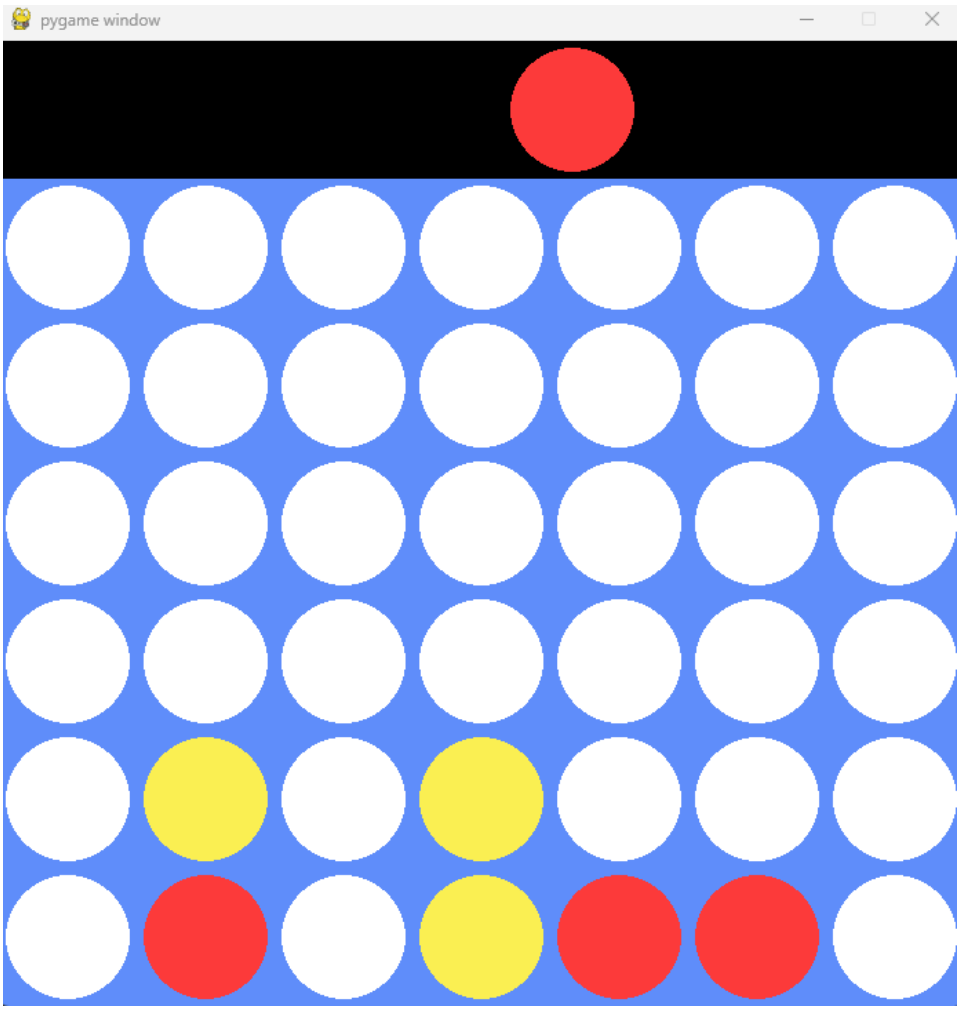
در این بخش (که اکثر آن مربوط به جزئیات پیاده سازی کتابخانه pygame می باشد و نیازی به توضیح نیست) در حلقه داخلی اول تا وقتی که دکمه خروج را نزدیم پنجره باز می ماند. در غیر این صورت تا زمانی که بازیکن موس را بر روی نوار بالا حرکت میدهد، یک دایره به رنگ مهره آن بازیکن در نوار بالا نمایان می شود و این صفحه مداوما آپدیت میشود. در نهایت به انتخاب بازیکن و جایی که بازیکن بر روی صفحه کلیک کرده است مهره بر روی صفحه بازی انداخته می شود (بعد از این ک حرکت مُجازی باشد و آن نقطه از صفحه خالی باشد). در انتها اگر بازیکن برنده شده باشد با تابع winning_move بررسی خواهد شد و روند کار در ادامه انجام خواهد شد. سپس عدد متغیر turn معکوس خواهد شد.

حال وقتی که بازیکن انتخاب خود را کرده است. در خارج حلقه بررسی خواهد شد که اگر بازی تمام نشده است و نوبت agent باشد، با استفاده از تابع minimax، ستون مورد نظر و مقدار heuristic مشخص شده را بر میگرداند.علت این که در تابع minimaxab عمق ۶ انتخاب شده است، این است که برنامه هرچقدر توانایی محاسبه در اعماق بالاتر را داشته باشد کاراتر بوده اما (با توجه به مقدار دهی دستی) اگر عمق بیشتر از ۶ شود درخت heuristic دارای تعداد بسیار زیادی شاخه خواهد شد و سرعت برنامه پایین می آید. بنابراین عمق ۶ بهترین عمق برای محاسبه می باشد. در پایان که agent انتخاب خود را کرده است بررسی می شود که آیا agent برنده بازی است یا خیر؛ اگر برنده بازی بود، بازی تمام می شود و اگر هنوز برنده نشده باشد نوبت عوض میشود

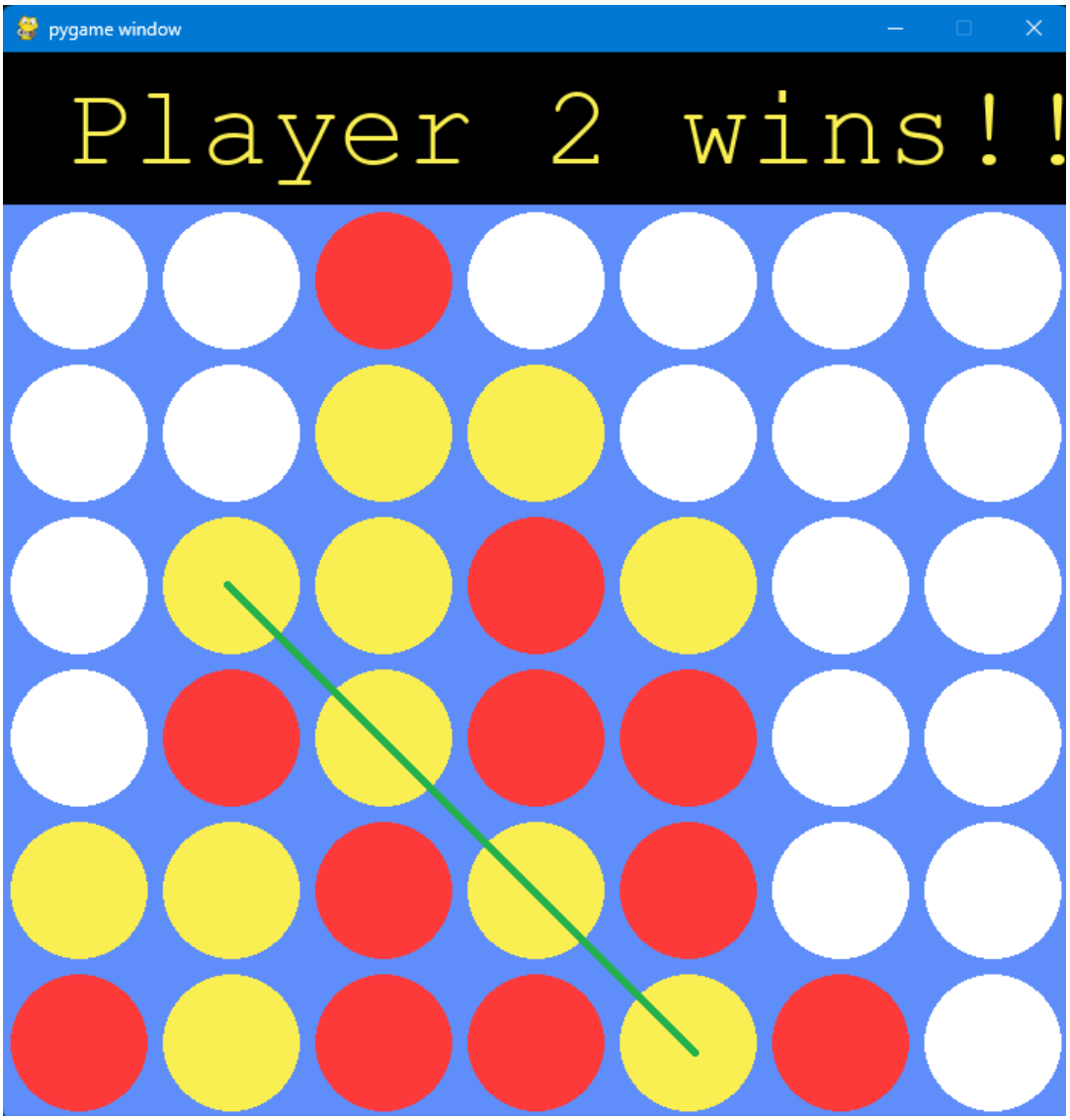
```
while not game_over:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

        if event.type == pygame.MOUSEMOTION:
            pygame.draw.rect(screen, BLACK, (0, 0, width, SQUARESIZE))
            posx = event.pos[0]
            if turn == PLAYER:
                pygame.draw.circle(screen, RED, (posx, int(SQUARESIZE/2)), RADIUS)

        pygame.display.update()

        if event.type == pygame.MOUSEBUTTONDOWN:
            pygame.draw.rect(screen, BLACK, (0, 0, width, SQUARESIZE))

            # Ask for Player 1 Input
            if turn == PLAYER:
                posx = event.pos[0]
                col = int(math.floor(posx / SQUARESIZE))

                if is_valid_location(board, col):
                    row = get_next_open_row(board, col)
                    drop_piece(board, row, col, PLAYER_PIECE)

                    if winning_move(board, PLAYER_PIECE):
                        label = myfont.render("Player 1 wins!!", 1, RED)
                        screen.blit(label, (40, 10))
                        game_over = True

                    turn = (turn + 1) % 2

                    draw_board(board)


    # Ask for Player 2 Input
    if turn == AI and not game_over:

        col, minimax_heuristic = minimax(board, 6, -math.inf, math.inf, True)

        if is_valid_location(board, col):
            row = get_next_open_row(board, col)
```

```
            drop_piece(board, row, col, AI_PIECE)
            if winning_move(board, AI_PIECE):
                label = myfont.render("Player 2 wins!!", 1, YELLOW)
                screen.blit(label, (40,10))
                game_over = True

            draw_board(board)

            turn = (turn + 1) % 2

    if game_over:
        pygame.time.wait(2000)
```

رابط گرافیکی بازی :



بازی بعد از برنده شدن agent :



جمع بندی : موارد قرمز امتیازی هستند

پیاده سازی منطق بازی (۲۰ امتیاز) : انجام شده. برنده شدن بازی با ردیف کردن ۴ مهره به صورت قطری(۱۰ امتیاز)

پیاده سازی رابط بازی (۱۰ امتیاز) : انجام شده. استفاده از کتابخانه های گرافیکی (۱۰ امتیاز)

پیاده سازی درخت مینی-مکس برای بازی (۳۰ امتیاز) : انجام شده

پیاده سازی الگوریتم minimax alpha beta pruning (۲۰ امتیاز) : انجام شده

به دست آوردن heuristic (۲۰ امتیاز) : انجام شده

گزارش (۵ امتیاز) : انجام شده

مجموع : ۱۲۵