Assignment 2

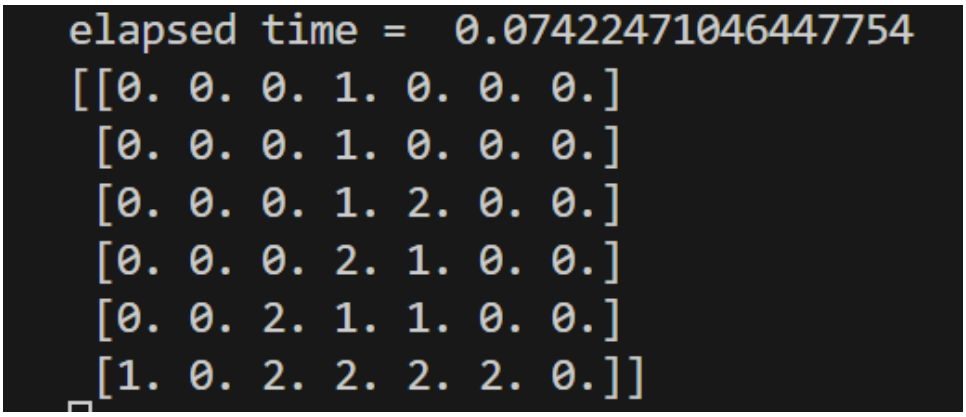
Youssef Mostafa Ghallab :7678

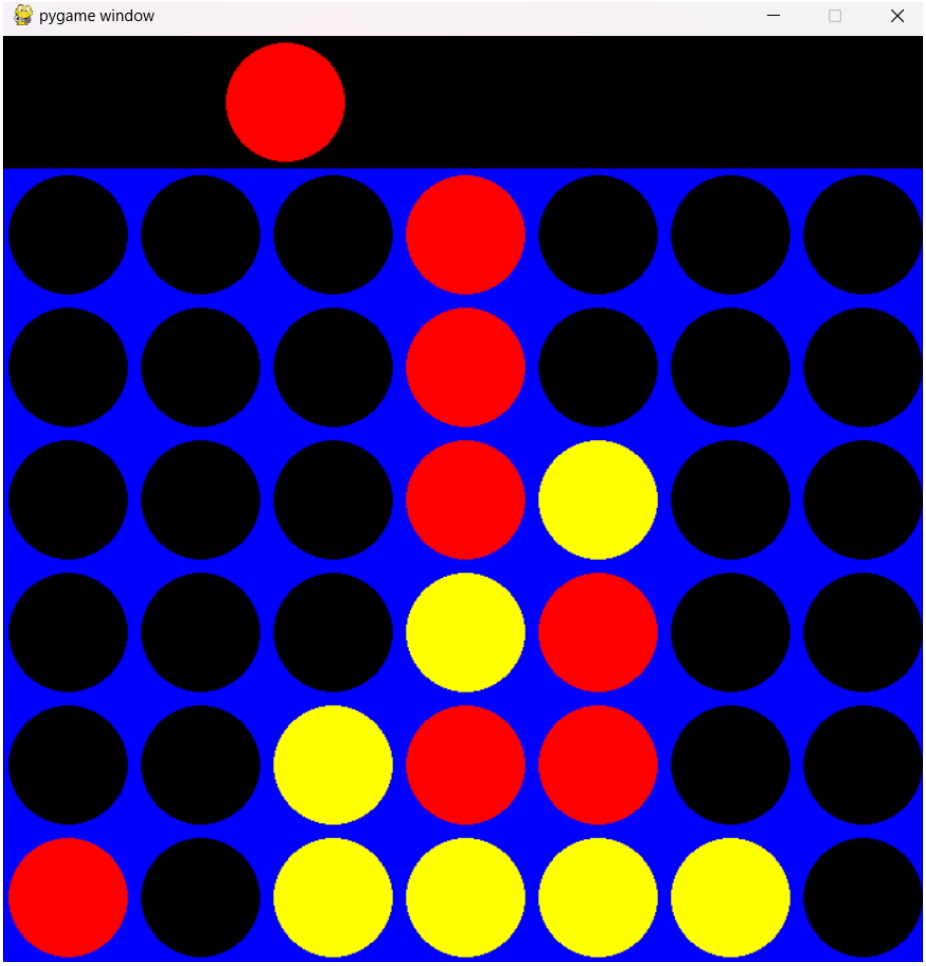
Osama Mostafa Mohamed: 7950

- Using Minimax Algorithm to Make Connect 4 AI Agent

1-sample runs

1-

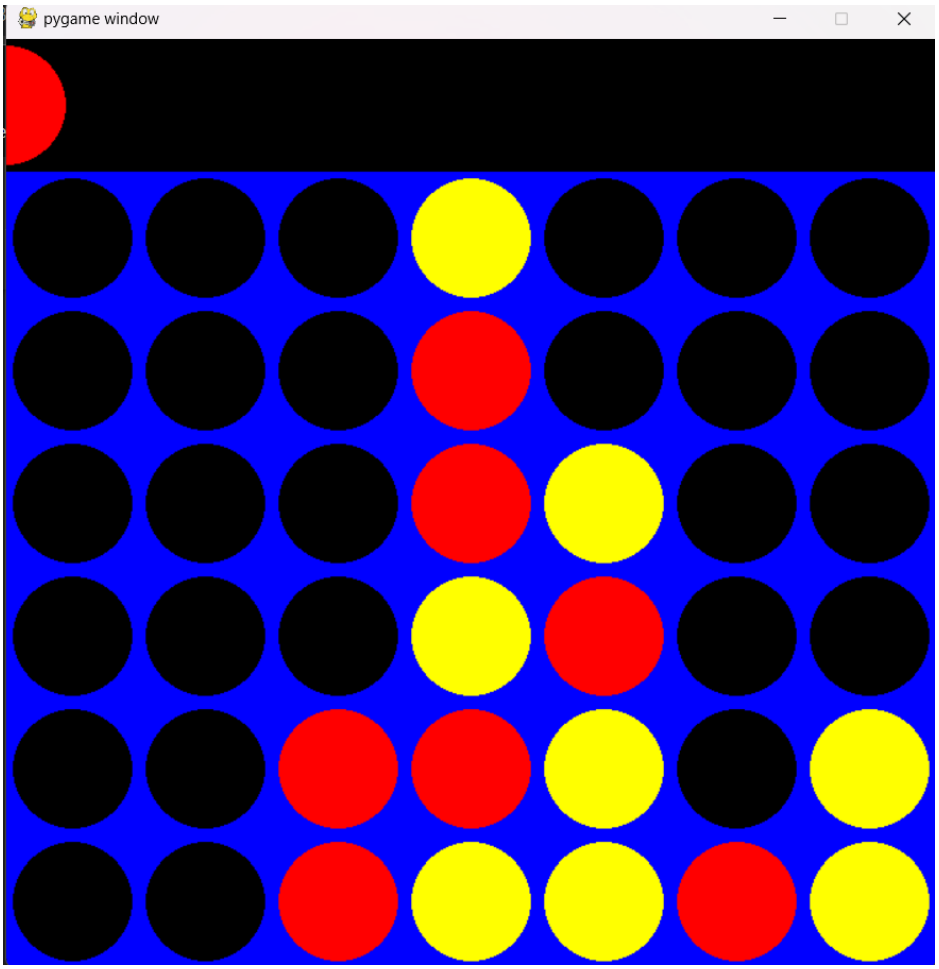



pygame window

```
minimizing player, depth = 3
465
465
elapsed time =  0.0010571479797363281
[[2. 1. 1. 2. 2. 2. 0.]
 [1. 2. 2. 2. 2. 1. 2.]
 [1. 1. 2. 2. 1. 2. 2.]
 [2. 2. 2. 2. 2. 1. 1.]
 [1. 1. 1. 2. 2. 1. 1.]
 [1. 1. 2. 1. 1. 1. 1.]]
[[2. 1. 1. 2. 2. 2. 1.]
 [1. 2. 2. 2. 2. 1. 2.]
 [1. 1. 2. 2. 1. 2. 2.]
 [2. 2. 2. 2. 2. 1. 1.]
 [1. 1. 1. 2. 2. 1. 1.]
```

2- sample (expecti_minimax)

```
elapsed time =  0.23761487007141113
[[0. 0. 0. 2. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 1. 2. 0. 0.]
 [0. 0. 0. 2. 1. 0. 0.]
 [0. 0. 1. 1. 2. 0. 2.]
 [0. 0. 1. 2. 2. 1. 2.]]
```



2-comparison between the 2 algorithms in terms of time taken and nodes expanded at different K values.

- 1st algorithm

-k =4

-time 0.07

-2 Nd algorithm

-k = 4

- time = 0.24

❖ Data structure used

Score: dictionary

Board: Numpy 2d array

❖ Algorithms:

```
def minimax_without_pruning(board, depth, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_full(board)

    if depth == 0 or is_terminal:
        return (None, score_position(board, AI_PIECE))

    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)
        print("maximizing player, depth = \n",depth)
        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_without_pruning(b_copy, depth-1, False)[1]
            if new_score > value:
```

```python
                value = new_score
                column = col
        return column, value

    else:
        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_without_pruning(b_copy, depth-1, True)[1]
            if new_score < value:
                value = new_score
                column = col
        return column, value



def minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal =  is_full(board)
    if depth == 0 or is_terminal:
            return (None, score_position(board, AI_PIECE))

    if maximizingPlayer:
        print("maximizing player,depth =\n",depth)
        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, alpha, beta, False)[1]
            print(new_score)

            if new_score > value:
                value = new_score
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else:
        print("minimizing player, depth =",depth)
        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, alpha, beta, True)[1]
            print (new_score)
            if new_score < value:
                value = new_score
                column = col
            beta = min(beta, value)
            if alpha >= beta:
```

```
                break
        return column, value


def expected_minimax(board, depth, alpha, beta, maximizingPlayer):
    valid_locations = get_valid_locations(board)
    is_terminal = is_full(board)
    if depth == 0 or is_terminal:
        return (None, score_position(board, AI_PIECE))

    if maximizingPlayer:
        value = -math.inf
        column = random.choice(valid_locations)  # Initial placeholder column

        for col in valid_locations:
            if is_valid_location(board, col):  # Check if column is not full
                row = get_next_open_row(board, col)
                b_copy = board.copy()
                drop_piece(b_copy, row, col, AI_PIECE)

                # Calculate expected value using separate minimax calls
                main_value = 0.6*expected_minimax(b_copy, depth - 1, alpha, beta,
False)[1]
                main_col = expected_minimax(b_copy, depth - 1, alpha, beta,
False)[0]
                left_value = right_value = 0
                right_neighbor_col = col + 1
                left_neighbor_col = col - 1
                # Check and call minimax for valid neighbors (corrected
probabilities for edge cases)
```

```
                if right_neighbor_col in valid_locations and left_neighbor_col
not in valid_locations:
                    right_b_copy = board.copy()
                    drop_piece(right_b_copy, get_next_open_row(right_b_copy,
right_neighbor_col), right_neighbor_col, AI_PIECE)
                    right_value = 0.4* expected_minimax(right_b_copy, depth - 1,
alpha, beta, False)[1]
                elif right_neighbor_col not in valid_locations and
left_neighbor_col in valid_locations: # Last column
                    left_neighbor_col = col - 1
                    left_b_copy = board.copy()
                    drop_piece(left_b_copy, get_next_open_row(left_b_copy,
left_neighbor_col), left_neighbor_col, AI_PIECE)
                    left_value = 0.4 * expected_minimax(left_b_copy, depth - 1,
alpha, beta, False)[1]
                elif right_neighbor_col in valid_locations and left_neighbor_col
in valid_locations:
                    left_neighbor_col = col - 1
                    left_b_copy = board.copy()
                    drop_piece(left_b_copy, get_next_open_row(left_b_copy,
left_neighbor_col), left_neighbor_col, AI_PIECE)
                    left_value = 0.2 * expected_minimax(left_b_copy, depth - 1,
alpha, beta, False)[1]
                    right_neighbor_col = col + 1
                    right_b_copy = board.copy()
                    drop_piece(right_b_copy, get_next_open_row(right_b_copy,
right_neighbor_col), right_neighbor_col, AI_PIECE)
                    right_value = 0.2 * expected_minimax(right_b_copy, depth - 1,
alpha, beta, False)[1]
                else:
                    main_value = main_value / 0.6

                expected_value = main_value + left_value + right_value
```

```
            if expected_value > value:
                value = expected_value
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break


    return column, value

else:  # Minimizing player (no change needed for expected minimax)
    value = math.inf
    column = random.choice(valid_locations)
    for col in valid_locations:
        if is_valid_location(board, col):  # Check if column is not full
            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, alpha, beta, True)[1]
            if new_score < value:
                value = new_score
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
    return column, value
```

❖ Assumption and necessary details

Heuristic functions:

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

    if window.count(piece) == 4:
        score += 45
    if window.count(piece) == 3 and window.count(EMPTY) == 1:
        score += 15
    if window.count(piece) == 2 and window.count(EMPTY) == 2:
        score += 7

    if window.count(opp_piece) == 3 and window.count(EMPTY) == 1:
        score -= 15
    if  window.count(opp_piece) == 2 and window.count(EMPTY) == 2:
        score -= 7
    if window.count(opp_piece) == 4:
        score -=45
```