Connect 6

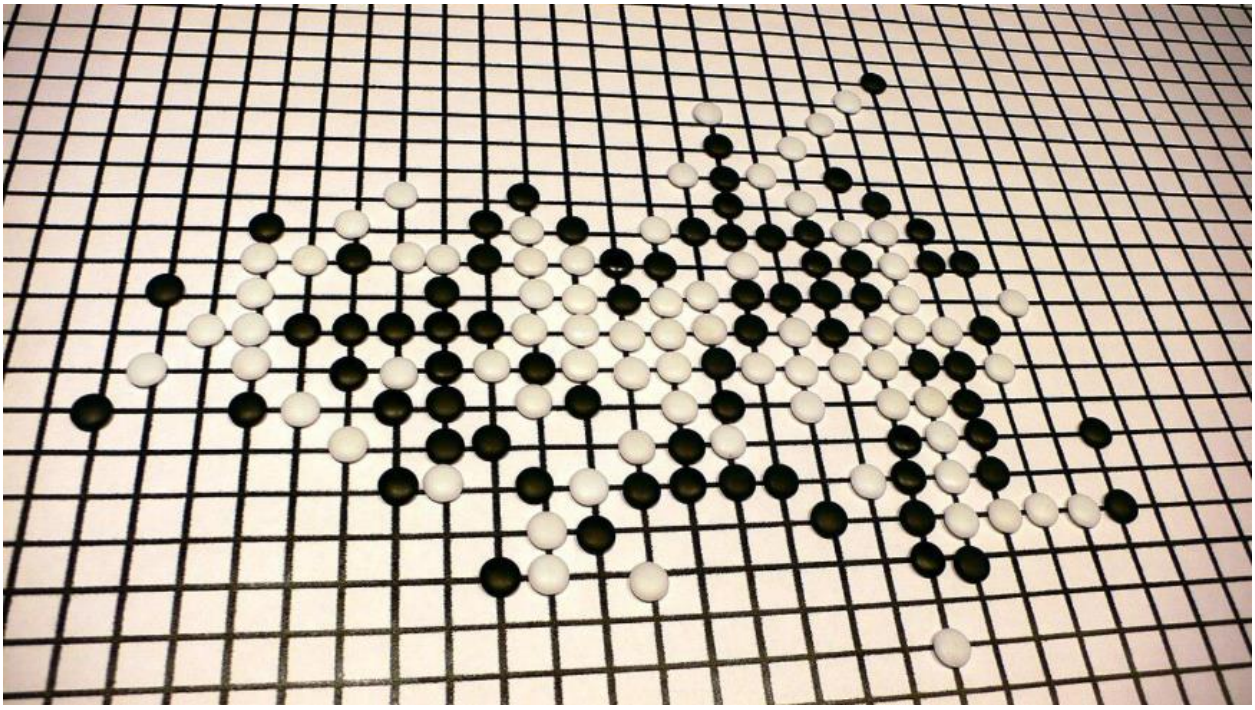
Teem members:

Hassan Abdelhamed Hassan Ebrahim	20210290	CS	3
Youssef Ahmed Abd Elmordy	20211058	CS	3
Toaa Assem Shendy Said	20210241	CS	3
Ranem Abo Elabas Mohamed Afify	20210336	CS	3
Omar Said Ahmed Abd Elfatah	20210604	CS	3
Mohamed Nasser Abo Elsouod Mohamed	20210840	CS	3
Abd Elrahman Ahmed Ebrahim Helmy	20210490	IS	3

1- Project idea:

An Intelligent Connect-6 Player using the Minimax Algorithm, Alpha-Beta Pruning, and Heuristic Functions.

Description: The project aims to develop an intelligent player for the Connect-6 game, an extended version of Connect Four, where players strive to connect six markers in a row horizontally, vertically, or diagonally on a 19x19 grid. This board game introduces increased complexity compared to Connect Four, making it a challenging problem for artificial intelligence (AI) systems to navigate strategically.

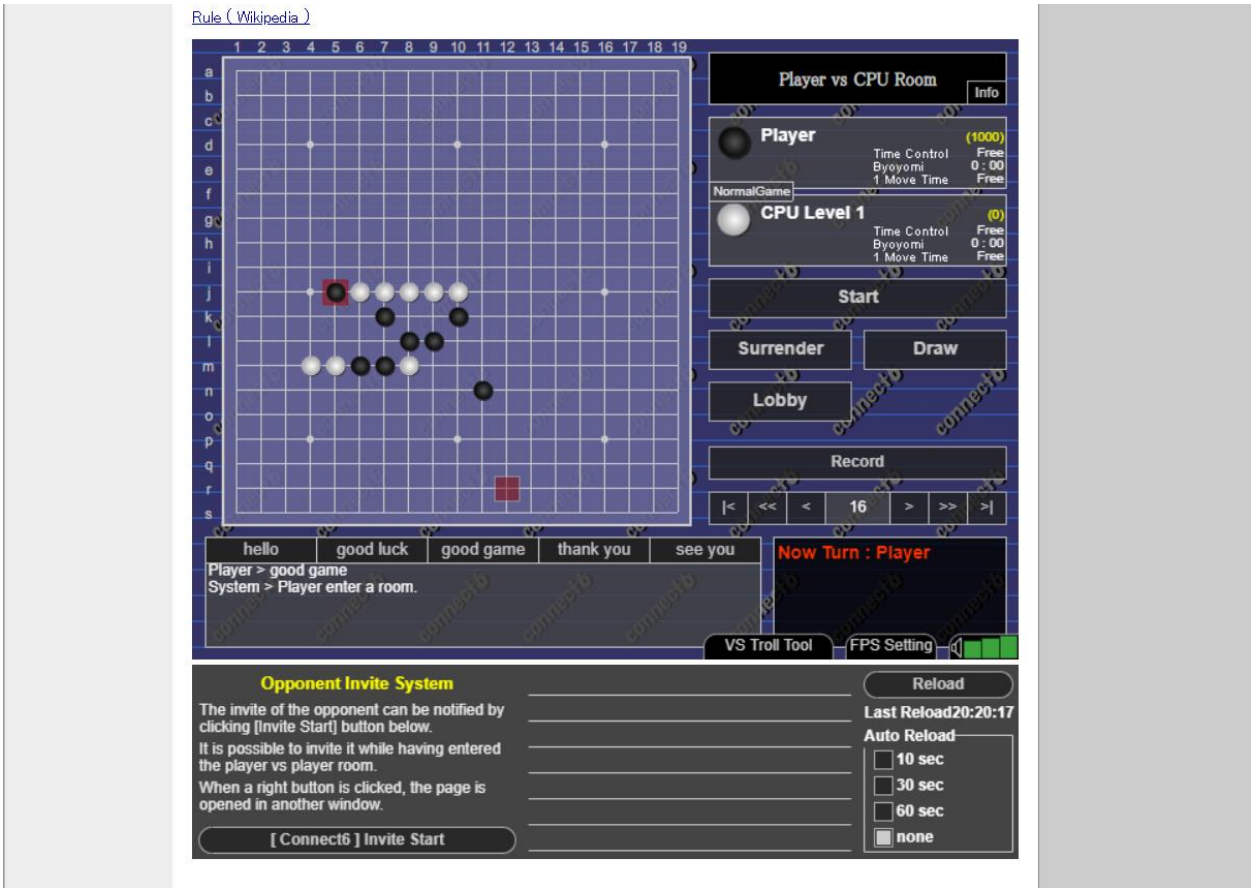


2- Similar_applications:

Connect 6 website.

Link: https://sdin.jp/en/browser/board/connect6

you can enjoy a player vs AI and player vs player. The first move is a black. The second move is a white. Whether it is a first move or it is the second mover is decided at random. Playable on PC, Smartphone etc. The name might remain for a little while (1 minutes at most) when the update of the page or the page is shut in the player vs player room.



3- An initial literature review of Academic publications (papers):

- Research Paper Link: https://en.m.wikipedia.org/wiki/Connect6

The rules of Connect6 are very simple and similar to the traditional game of Gomoku:

Players and stones: There are two players. Black plays first, and White second. Each player plays with an appropriate color of stones, as in Go and Gomoku.

Game board: Connect6 is played on a square board made up of orthogonal lines, with each intersection capable of holding one stone. In theory, the game board can be any finite size from 1×1 up (integers only), or it could be of infinite size. However, boards that are too small may lack strategy (boards smaller than 6×6 are automatic draws), and extremely large or infinite boards are of little practical use. 19×19 Go boards might be the most convenient. For a longer and more

challenging game, another suggested size is 59×59, or nine Go boards tiled in a larger square (using the join lines between the boards as additional grid lines).

Game moves: Black plays first, putting one black stone on one intersection. Subsequently, White and Black take turns, placing two stones on two different unoccupied spaces each turn.

Winner: The player who is the first to get six or more stones in a row (horizontally, vertically, or diagonally) wins. (This is a departure from Gomoku, where it must be exactly five in a row.)

According to Professor Wu, the handicap of black's only being able to play one stone on the first turn means that the game is comparatively fair; unlike similar games such as Gomoku and Connect Four, which have been proven to give the first player a large advantage, possibly no additional compensation is necessary to make the game fair

- Research Paper Link: https://senseis.xmp.net/?Connect6

Connect 6

Connect6 was introduced by Professor I-Chen Wu at Department of Computer Science and Information Engineering, National Chiao Tung University, is a two-player game similar to Gomoku.

Connect6 was designed as complex and hard to brute-force by computers. It have tree complexity of ≈10^172 (GO at 19x19 board have in comparsion tree complexity of ≈10^171.

Two players, Black and White, alternately place two stones of their own colour, Black and White respectively, on empty intersections of a Go-like board, except that Black (the first player) places one stone only for the first move. The one who gets six or more stones in a row (horizontally, vertically or diagonally) first wins the game.

The handicap of black's only being able to play one stone on the first turn means that the game is comparatively fair; unlike similar games such as Gomoku has been proven to give the first player a large advantage, possibly no additional compensation is necessary to make the game fair.

Game Rules

The rules of Connect6 are very simple and similar to the traditional game of Gomoku:

Players and stones: There are two players. Black plays first, and White second. Each player plays with an appropriate color of stones, as in Go and Gomoku.

Game board: Connect6 is played on a square board made up of orthogonal lines, with each intersection capable of holding one stone. In theory, the game board can be any finite size from 1×1 up (integers only), or it could be of infinite size. However, boards that are too small may lack

strategy (boards smaller than 6×6 are automatic draws), and extremely large or infinite boards are of little practical use. 19×19 Go boards might be the most convenient for beginners.

Professional games are played on 59x59 boards 9 gobans tiled in a square.

Game moves: Black plays first, putting one black stone on one intersection. Subsequently, White and Black take turns, placing two stones on two different unoccupied spaces each turn.

Winner: The player who is the first to get six or more stones in a row (horizontally, vertically, or diagonally) wins. (This is a departure from Gomoku, where it must be exactly five in a row.)

- Research Paper Link: https://www.chessprogramming.org/Connect6

Connect6, Connect(m,n,6,2,1)

a two-player abstract strategy board game of the k-in-a-row family similar to Gomoku, introduced in 2003 by I-Chen Wu and presented at Advances in Computer Games 11 in 2005. Black and White alternately place two stones of their own colour on empty intersections of a Go-like board, except that Black (the first player) places one stone only for the first move [2]. The one who gets six or more stones in a row (horizontally, vertically or diagonally) first wins the game. Most often, Connect6 is played on a 19x19 Go board, proposed for professional players is a 59x59 board [3]. Since 2006, Connect6 is played regularely by computers at the Computer Olympiad organized by the ICGA [4]. Search algorithms used are Alpha-Beta / MTD(f) along with VCF search (Victory by Continuous Four) to find a path to win in the endgame [5], and Monte-Carlo Tree Search, UCT, as well as Proof-Number Search also in conjunction with the novel relevance-zone-oriented proof (RZOP) search used to solve various openings, such as the Mickey Mouse opening [6].

- Research Paper Link: https://www.researchgate.net/publication/302199506_A_Case_study_of_Connect6_Game_FPGA-based_Implementation_Using_the_Multi-turn_Prediction_Algorithm

The Connect6 is a fair and highly complex game withsimple rules" [1]. Two players, with dark and light stones,alternately place two stones on empty intersections of a 19x19GO board for a turn. The initial turn is special because of thedark side always plays first and put one stone in the initialturn. The winner is the one who gets six stones in-a-row first(i.e. horizontally, vertically

or diagonally). Since 2003, manyConnect6 tournaments have been organized around the world[2]-[4].Although the rule is simple, a computer cannot solve allpossible situations due to a number of intensive computa-tional processes. For this reason, many different Connect6solutions are implemented in software and/or hardware. Jun-jie Tao et al. [5] proposed a method to construct an openingbook that ”contains an ocean of grandmasters game recordsor the excellent positions produced between computers andgrandmasters”, while I-Chen Wu et al. [6] presented a gridcomputing environment for Connect6 applications. However,those methods consume high cost and high power due to thepowerful computers. To avoid utilizing computers, KentaroSano et al. [7] designed a hardware and software co-designof a Connect6 AI with scalable streaming cores in an FPGA.Although the latency was improved, it was too far to reach thetime limit of almost Connect6 contests (i.e. 1 second for eachturn). Takahio Watanabe et al. [8] presented an approach ofhardware accelerator implementation in an FPGA using two-stage pipelined evaluation. However, the logic utilization istoo large to implement in such a low-cost FPGA prototype.In addition, the others experimental results and analysis ofthe systems are not discussed.

- Research Paper Link: https://www.scaler.com/topics/artificial-intelligence-tutorial/min-max-algorithm/

The minimax algorithm in game theory helps the players in two-player games decide their best move. It is crucial to assume that the other player is also making the best move while determining the best course of action for the present player. Each player attempts to reduce the maximum loss that they can suffer in the game. This article will cover the minimax algorithm's concept, its working, its properties, and other relevant ideas.

Working of Min-Max Algorithm

The Min Max algorithm recursively evaluates all possible moves the current player and the opponent player can make. It starts at the root of the game tree and applies the MinMax algorithm to each child node. At each level of the tree, the algorithm alternates between maximizing and minimizing the node's value. The player who will be winning the game is the maximizing player, whereas the player who will be losing the game is the minimizing player. The maximizing player chooses the child node with the highest value, while the minimizing player chooses the child node with the lowest value. This move is considered the optimal move for the player. The algorithm evaluates the child nodes until it reaches a terminal node or a predefined depth. When it reaches a terminal node, the algorithm returns the heuristic value of the node. The heuristic value is a score that represents the value of the game's current state.

4- Main Functions:

- get_board_size()

  Prompts the user to enter the desired board size using a GUI.

  Ensures that the input is a valid integer greater than or equal to 6.

```
def get_board_size():
    while True:
        try:
            board_size = simpledialog.askinteger("Board Size", "Enter the desired board size (n):")
            if board_size is None:
                # User pressed Cancel
                print("Board size input canceled. Exiting.")
                exit()
            if board_size < 6:
                print("Board size must be at least 6. Please enter a valid size.")
            else:
                return board_size
        except ValueError:
            print("Invalid input. Please enter a valid integer.")
```

- create_board(n)

  Creates a Connect Six game board with dimensions n x n.

```
def create_board(n):
    board = np.zeros((n, n))
    return board


'''
```

- drop_piece(board, row, col, piece)

  Places a game piece on the game board at the specified row and column.

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

- is_valid_location(board, col)

  Checks if the specified column is a valid location for placing a game piece.

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


'''
```

- get_next_open_row(board, col)

  Finds the next available row in the specified column.

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

- print_board(board)

  Prints the Connect Six game board to the console.

```
'''
def print_board(board):
    print(np.flip(board, 0))


'''
```

- winning_move(board, piece)
  Checks for a winning move on the game board for a specified player or AI.

```
def winning_move(board, piece):
    # Check horizontal locations for win
    for c in range(board.shape[1] - WINDOW_LENGTH + 1):
        for r in range(board.shape[0]):
            if all(board[r][c + i] == piece for i in range(WINDOW_LENGTH)):
                return True

    # Check vertical locations for win
    for c in range(board.shape[1]):
        for r in range(board.shape[0] - WINDOW_LENGTH + 1):
            if all(board[r + i][c] == piece for i in range(WINDOW_LENGTH)):
                return True

    # Check positively sloped diagonals
    for c in range(board.shape[1] - WINDOW_LENGTH + 1):
        for r in range(board.shape[0] - WINDOW_LENGTH + 1):
            if all(board[r + i][c + i] == piece for i in range(WINDOW_LENGTH)):
                return True

    # Check negatively sloped diagonals
    for c in range(board.shape[1] - WINDOW_LENGTH + 1):
        for r in range(WINDOW_LENGTH - 1, board.shape[0]):
            if all(board[r - i][c + i] == piece for i in range(WINDOW_LENGTH)):
                return True
```

- evaluate_window(window, piece)
  Evaluates the score of a window of game pieces for a specified player or AI.

- score_position(board, piece)
  Evaluates the overall score of the game board for a specified player or AI.

```python
# heuristic function
def evaluate_window(window, piece):
    score = 0
    opp_piece = PLAYER_PIECE if piece == AI_PIECE else AI_PIECE

    # Add more factors to the evaluation
    score += window.count(piece) * 10
    score -= window.count(opp_piece) * 20
    score += window.count(EMPTY) * 10

    return score

# heuristic function
def score_position(board, piece):
    score = 0

    # Score center column
    center_array = [int(i) for i in list(board[:, board.shape[1] // 2])]
    center_count = center_array.count(piece)
    score += center_count * 300

    # Score Horizontal
    for r in range(board.shape[0]):
        row_array = [int(i) for i in list(board[r, :])]
        for c in range(board.shape[1] - WINDOW_LENGTH + 1):
            window = row_array[c:c + WINDOW_LENGTH]
            score += evaluate_window(window, piece)
```

```python
    # Score Vertical
    for c in range(board.shape[1]):
        col_array = [int(i) for i in list(board[:, c])]
        for r in range(board.shape[0] - WINDOW_LENGTH + 1):
            window = col_array[r:r + WINDOW_LENGTH]
            score += evaluate_window(window, piece)

    # Score positively sloped diagonal
    for r in range(board.shape[0] - WINDOW_LENGTH + 1):
        for c in range(board.shape[1] - WINDOW_LENGTH + 1):
            window = [board[r + i][c + i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    # Score negatively sloped diagonal
    for r in range(board.shape[0] - WINDOW_LENGTH + 1):
        for c in range(WINDOW_LENGTH - 1, board.shape[1]):
            window = [board[r + 5 - i][c - i] for i in range(WINDOW_LENGTH)]
            score += evaluate_window(window, piece)

    return score
```

- is_terminal_node(board)
  Checks if the game has reached a terminal state (win, lose, or draw).

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


'''
```

- minimax(board, depth, alpha, beta, maximizingPlayer)
  Executes the Minimax algorithm to find the best move for the AI player.

```
def minimax(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, -10000000000000)
            else: # Game is over, no more valid moves
                return (None, 0)
        else: # Depth is zero
            return (None, score_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_score = minimax(b_copy, depth-1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                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_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
```

- alpha_beta_pruning(board, depth, alpha, beta, maximizingPlayer, player_turn)
  Executes the Alpha-beta algorithm to find the best move for the AI player.

```python
def alpha_beta_pruning(board, depth, alpha, beta, maximizingPlayer, player_turn):
    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, -10000000000000
            else:  # Game is over, no more valid moves
                return None, 0
        else:  # Depth is zero
            if player_turn:  # Player's turn
                return None, score_position(board, PLAYER_PIECE)
            else:  # AI's turn
                return None, score_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_score = alpha_beta_pruning(b_copy, depth - 1, alpha, beta, False, player_turn)[1]
            if new_score > value:
                value = new_score
                column = col
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value
```

- get_valid_locations(board)
  Finds valid locations (columns) for placing a game piece on the game board.

```python
def get_valid_locations(board):
    valid_locations = []
    for col in range(board.shape[1]):
        if is_valid_location(board, col):
            valid_locations.append(col)
    return valid_locations
```

- pick_best_move(board, piece)
  Picks the best move for a specified player or AI using the Minimax algorithm.

```python
def pick_best_move(board, piece):
    valid_locations = get_valid_locations(board)
    best_score = -10000
    best_col = random.choice(valid_locations)

    for col in valid_locations:
        row = get_next_open_row(board, col)
        temp_board = board.copy()
        drop_piece(temp_board, row, col, piece)

        # Increase search depth for a stronger AI
        _, score = minimax(temp_board, 6, -math.inf, math.inf, False)

        if score > best_score:
            best_score = score
            best_col = col

    return best_col
```

- draw_board(board)
  Draws the Connect Six game board on the Pygame screen.

```python
def draw_board(board):
    # Fill the entire screen with white color
    screen.fill(WHITE)

    # Dynamically adjust square size based on the board size
    square_size = int(min(SQUARESIZE - 5, height / board.shape[0]))

    for c in range(board.shape[1]):
        for r in range(board.shape[0]):
            pygame.draw.rect(screen, GRAY, (c * SQUARESIZE, r * SQUARESIZE + SQUARESIZE, SQUARESIZE, SQUARESIZE), 5)

    for c in range(board.shape[1]):
        for r in range(board.shape[0]):
            if board[r][c] == PLAYER_PIECE:
                pygame.draw.circle(screen, RED, (int(c * SQUARESIZE + SQUARESIZE / 2),
                                                 height - int(r * SQUARESIZE + SQUARESIZE / 2)), square_size // 2)
            elif board[r][c] == AI_PIECE:
                pygame.draw.circle(screen, YELLOW, (int(c * SQUARESIZE + SQUARESIZE / 2),
                                                    height - int(r * SQUARESIZE + SQUARESIZE / 2)), square_size // 2)

    pygame.display.update()
```

- Main Game Loop
  Implements the main game loop, handling player and AI turns, user input, and updating the game display.
  Integrates the functions mentioned above to create the Connect Six game.

```
# Main game loop
while not game_over:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            game_over = True  # Set the game_over flag to exit the loop


        pygame.display.update()

        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.type == pygame.MOUSEBUTTONDOWN:
#                 pygame.draw.rect(screen, BLACK, (0, 0, width, SQUARESIZE)
                posx, posy = event.pos
                col = int((posx ) // SQUARESIZE)
                row = int((posy ) // SQUARESIZE)
                num_rows, num_cols = board_size, board_size
                row = num_rows - 1 - row
                print(f"Row: {row}, Column: {col}")

                if is_valid_location(board, col):
                    drop_piece(board, row, col, PLAYER_PIECE)
                    print_board(board)
                    draw_board(board)

                    if winning_move(board, PLAYER_PIECE):
                        print("Player 1 wins!!")
                        label = myfont.render("Player 1 wins!!", 1, RED)
                        screen.blit(label, (40, 10))
                        game_over = True
                    elif winning_move(board, AI_PIECE):
                        print("AI wins!!")
                        label = myfont.render("Ai wins!!", 1, YELLOW)
                        screen.blit(label, (40, 10))
                        game_over = True
```

```
                elif len(get_valid_locations(board)) == 0:
                    print("It's a draw!")
                    label = myfont.render("It's a draw!", 1, (255, 255, 255))
                    screen.blit(label, (40, 10))
                    game_over = True

                turn += 1
                turn = turn % 2



if turn == AI and not game_over:
    col, minimax_score = minimax(board, 4, -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)
        print_board(board)
        draw_board(board)

    if winning_move(board, AI_PIECE):
            label = myfont.render("AI wins!!", 1, YELLOW)
            screen.blit(label, (40, 10))
            game_over = True
    elif len(get_valid_locations(board)) == 0:
            label = myfont.render("It's a draw!", 1, (255, 255, 255))
            screen.blit(label, (40, 10))
            game_over = True



    turn += 1
    turn = turn % 2
```

```
    pygame.display.update()

    if game_over:
        pygame.time.delay(2000)  # Increased delay to 3 seconds

# Quit pygame and exit the program
pygame.quit()
sys.exit()
```

5- Details of Algorithm:

Analysis of Results:

Effectiveness of Minimax and Alpha-Beta Pruning:

Evaluate the success rate of the Minimax and Alpha-Beta Pruning algorithms in playing Connect6. This includes the ability to find optimal moves and win the game against opponents.

Computation Time:

Analyze the computational efficiency of Alpha-Beta Pruning compared to standard Minimax. Assess whether Alpha-Beta Pruning significantly reduces the search space, leading to faster decision-making without sacrificing accuracy.

Scalability:

Test the scalability of the algorithms by varying the size of the Connect6 board. Assess whether the algorithms can handle larger boards effectively without a significant increase in computation time.

Advantages / Disadvantages:

Minimax Algorithm:

Advantages:

Guarantees finding the optimal move in terms of maximizing the chance of winning or minimizing the chance of losing.

Provides a systematic approach to exploring the entire game tree.

Disadvantages:

Computationally expensive, especially for large game trees.

Not practical for deep exploration without optimizations.

Alpha-Beta Pruning:

Advantages:

Significantly reduces the number of nodes explored, leading to faster decision-making.

Improves the efficiency of the Minimax algorithm without compromising the optimality of the solution.

Disadvantages:

May not provide a speedup in scenarios where the game tree is already relatively small.

Implementation complexity compared to basic Minimax.

Why Did the Algorithm Behave in Such a Way?

Optimal Move Selection:

Minimax ensures the selection of optimal moves by considering all possible outcomes. However, the efficiency of this process heavily depends on the depth of the search tree.

Alpha-Beta Pruning Efficiency:

Alpha-Beta Pruning behaves more efficiently by pruning branches that cannot affect the final decision. The behavior is influenced by the effectiveness of pruning in reducing the search space.

Future Modifications:

Enhanced Heuristics:

Improve the evaluation function with more sophisticated heuristics. This can lead to a better estimation of board positions, enabling the algorithm to make more informed decisions.

Adaptive Depth Limit:

Implement an adaptive depth limit for Minimax based on the current game state and available computational resources. This can optimize the algorithm's performance for different scenarios.

Parallelization:

Explore parallelization techniques to further improve the computational efficiency, especially for large game trees. Distributing the workload across multiple processors or threads can lead to faster decision-making.

Exploration of Alternative Algorithms:

Consider exploring alternative algorithms, such as Monte Carlo Tree Search (MCTS), and assess their performance compared to Minimax with Alpha-Beta Pruning. MCTS can handle larger search spaces effectively.

Machine Learning Integration:

Integrate machine learning techniques to enhance the AI's decision-making. Reinforcement learning or neural network-based evaluation functions could improve the AI's ability to generalize and adapt to various game scenarios.

Experiment with Different Evaluation Functions:

Experiment with different evaluation functions and weights for features. Fine-tune these parameters to find the optimal balance between simplicity and accuracy in evaluating board positions.

By addressing these considerations and experimenting with modifications, you can aim to enhance the performance of the Minimax with Alpha-Beta Pruning algorithms in playing Connect6. Keep in mind that the success of modifications will depend on the specific characteristics of the Connect6 game and the goals of the AI development.

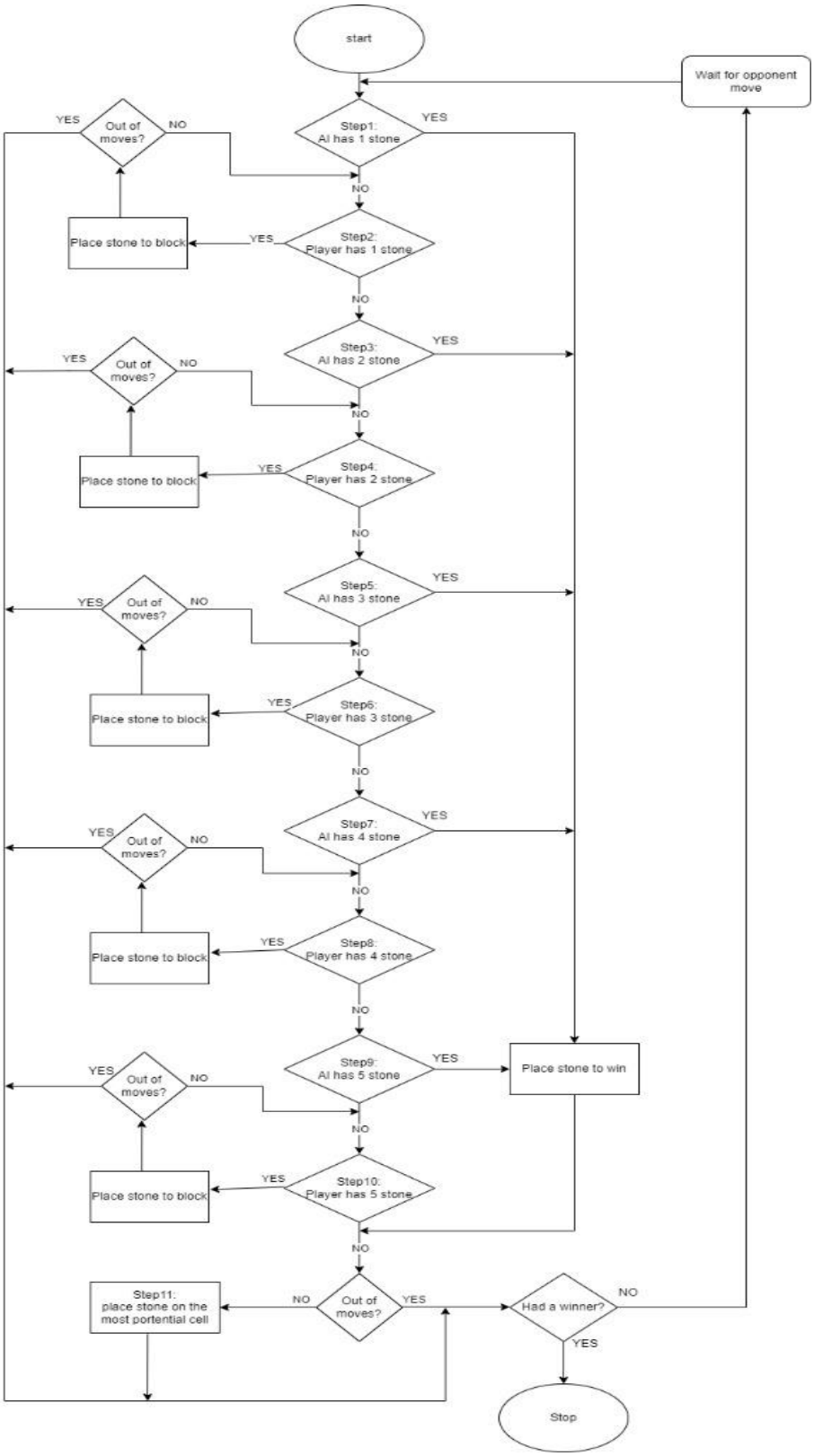
6- Development platform:

all packages used in the project: pygame, math, random, matplotlib.pyplot coding

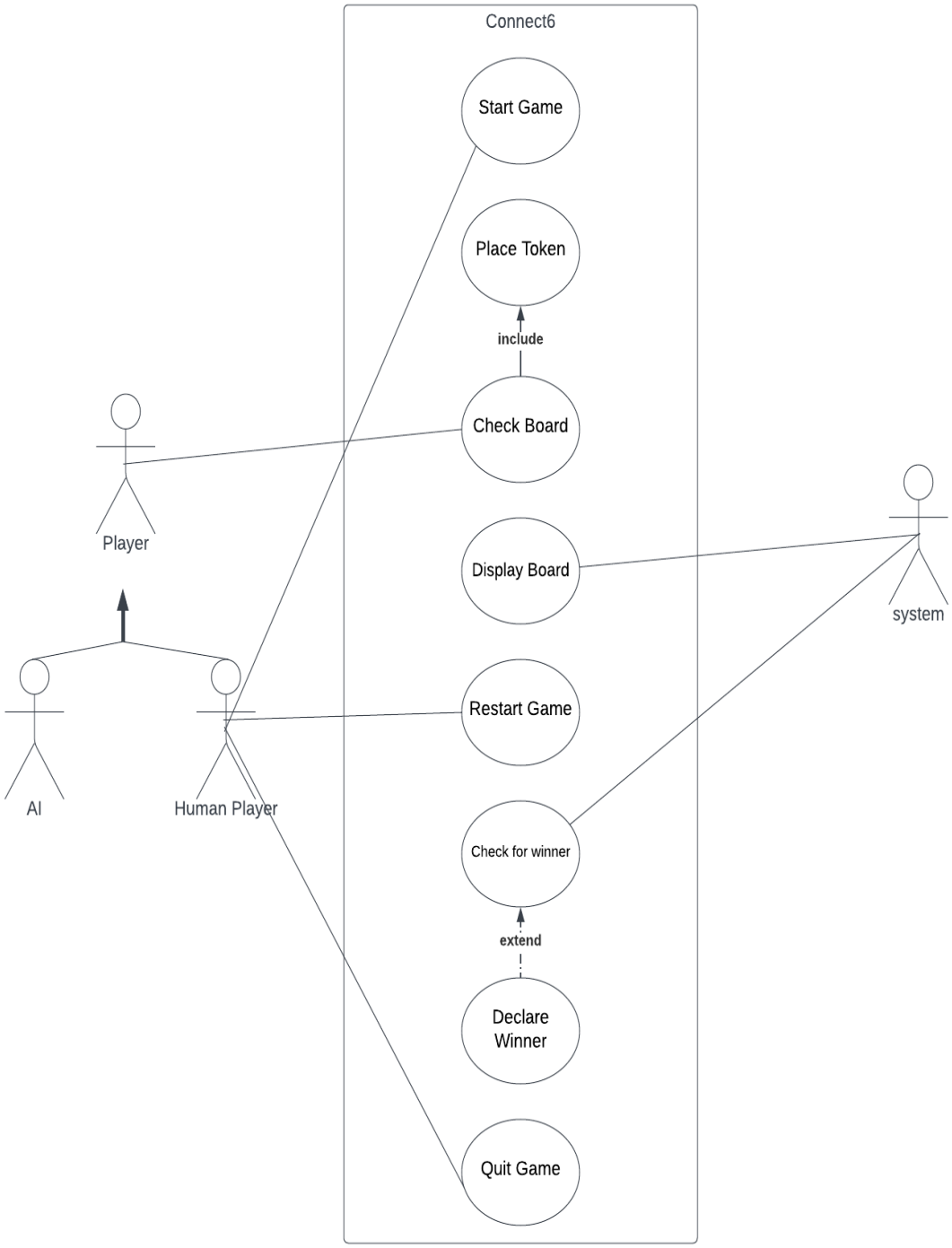
platforms: spyder and pycharm language: python 3.9

7- Diagrams:

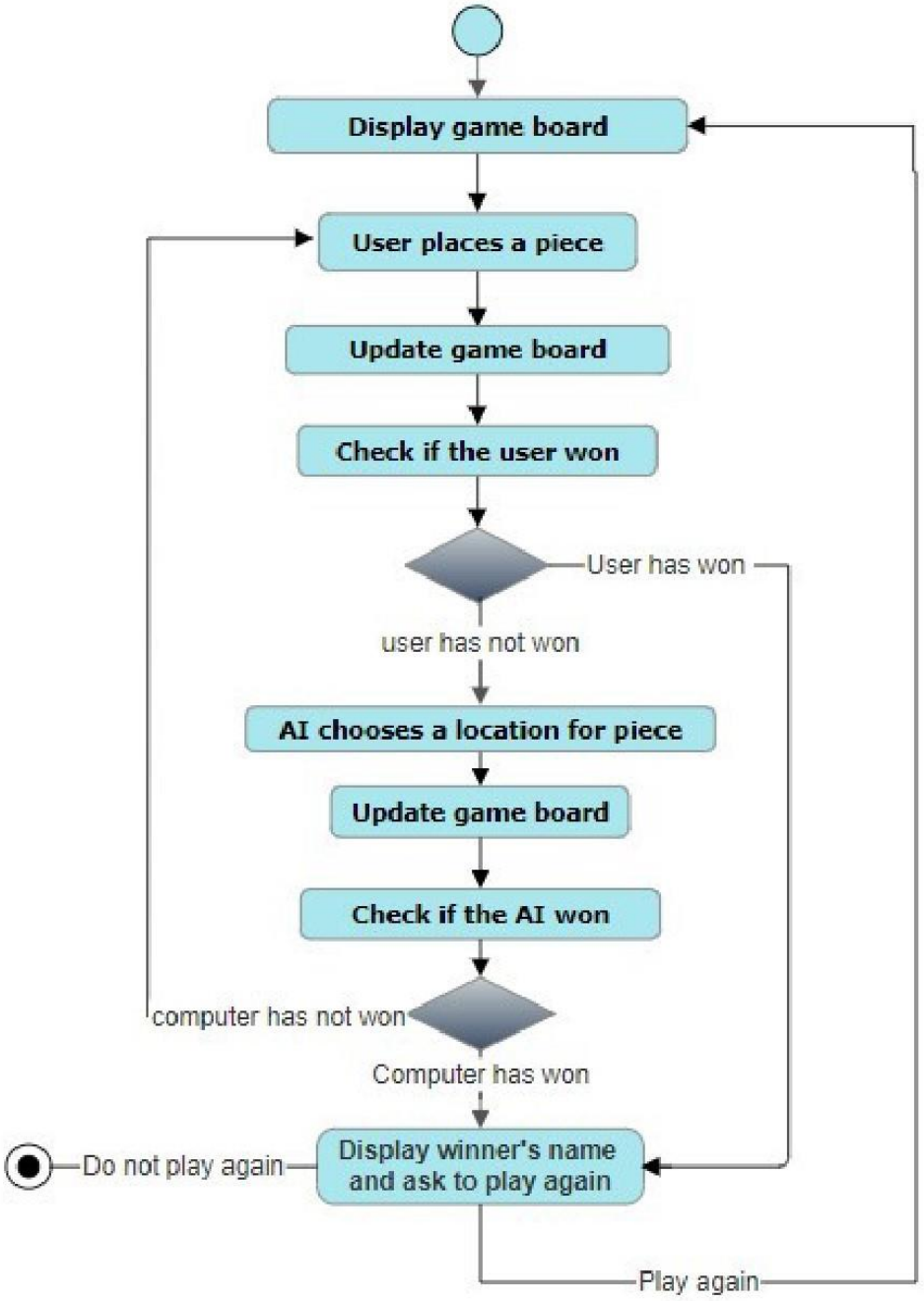
Flow chart:



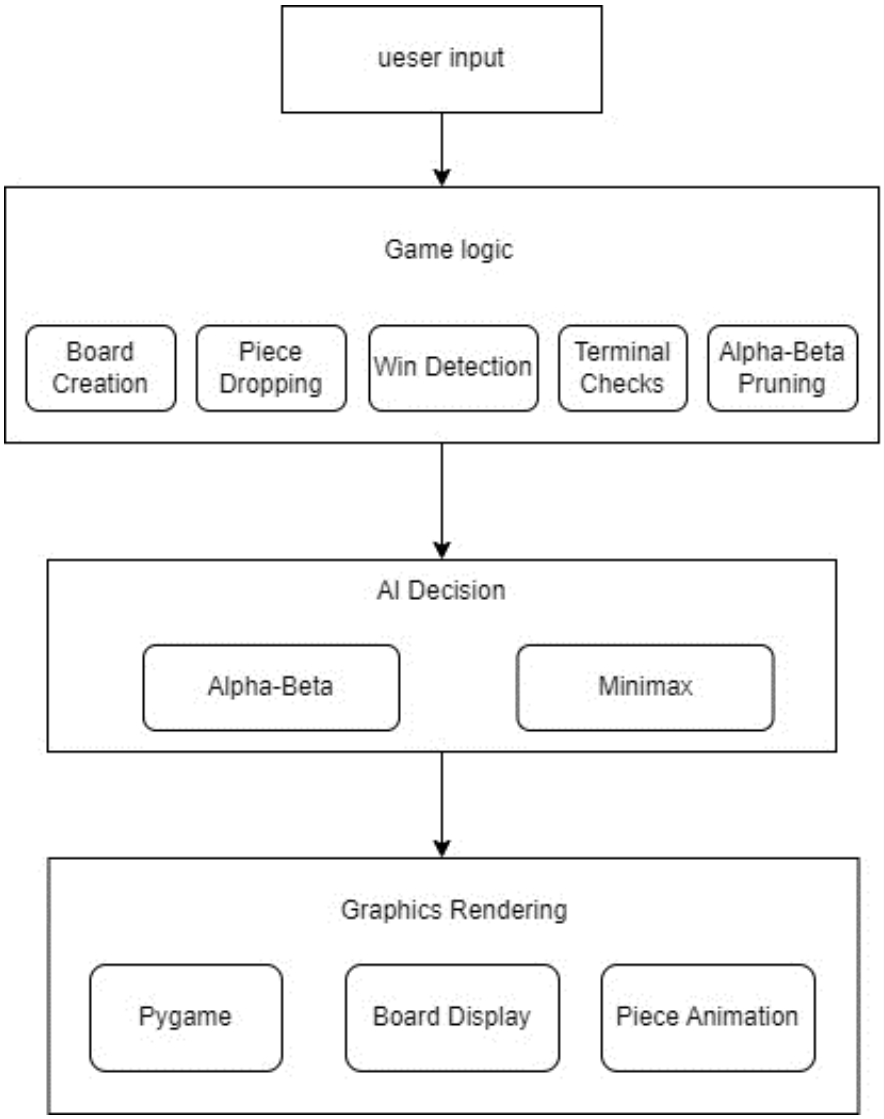
Use case :


Connect6
Start Game
Place Token
include
Check Board
Display Board
Restart Game
Check for winner
extend
Declare Winner
Quit Game
Player
AI
Human Player
system

Activity diagram:



Block diagram:



8- Main Functions:

Shared code :

https://github.com/HassanAbdelhamed22/Connect6_Project/tree/main

Thank You ☺