Final Project: Gomoku AI Game

Minerva University

CS152 - Harnessing Artificial Intelligence Algorithms

Prof. Shekhar

April 17, 2023

I. Problem Definition

Gomoku (also known as five in a row) is an abstract strategy board game that is normally played on a 15x15 grid board. The game is played by two players with one player using black and the other using white stones. However, there might be certain variations (such as it can be played on a 19x19 grid with other pieces and by more than two players). While starting the same, the board would be empty. Each player then takes turns placing their stone on the board. The player who win is the person who manages to place a sequence of five stones either in a row, horizontally, vertically, or diagonally line. If the board is full (meaning there are 225 stones on the board), but there is no sequence of length five, then it would be a draw.

Even though the rule of the game is relatively simple, it has a high branching factor meaning that for each turn there are many next possible moves that the other player can make (we can think that if the board state for each turn represents a node there are many child nodes that can be generated). To avoid losing, each player must be able to balance defense and offense efficiently. Or in other words, the player needs to make sure they are creating a sequence of five stones to win while blocking the opponent from doing so to prevent losing. To effectively do so, the players must be able to think ahead and explore all the next possible moves and outcomes to determine which move they would play next to optimize their chances of winning.

However, keeping track of their moves and predicting their opponent's moves to combat and play optimally is a challenging task. Therefore, in this project, I will create a Gomoku AI using a minimax alpha-beta pruning algorithm to help humans determine the best possible moves given a board state and can also play against humans to challenge them out. The game has been tested to be able to beat humans or give a draw in the game. The details of the algorithm implementation will be carried out below.

II. Solution Specification.

a. Minimax Search Algorithm

The game's characteristics is multiagent (2 agents specifically who take turns), deterministic, zero-sum, and fully observable (perfect information), which makes it suitable for the minimax algorithm.

The optimal solution is determined through the utility value of the final state of the game. The minimax search algorithm (using depth-first search) would try to find the optimal move for the max player by testing all possible paths and choosing the path that would result in the largest utility value in the terminal node assuming both players play optimally. On the other hand, the min player would want to get to a state with the lowest utility. We can see that it is a recursive algorithm, as for each state, it will continue to call the search function to go down one depth in the tree until it reaches way down the terminal nodes and backs up the minimax values with recursion. The recursive nature of the minimax algorithm becomes apparent as it traverses the game tree by repeatedly calling the search function to explore one level deeper until it reaches the terminal nodes. Once it goes to the end of the game, the algorithm backtracks up the tree, passing along values to determine the optimal move for the current player. The code for using the minimax algorithm (with alpha-beta pruning) and building the AI agent in the game is commented on thoroughly in the appendix.

b. Implementing alpha-beta pruning and depth cut-off.

However, as the search space and the number of game states increase exponentially with depth, it would be nearly impossible to develop the whole search tree. Therefore, we will implement several techniques to limit the scope of examining every state. First, we will use alpha-beta pruning to eliminate irrelevant subtrees that make no difference to the final outcome. .However, due to computation time and space constraints, even with alpha-beta pruning, we can not fully compute the utility in the terminal nodes (states where the game ends). Therefore, we would need to cut off the search early by giving the game tree max depth and using a heuristic evaluation function to estimate the expected utility. The heuristic function I used is a linear weighted sum function in which the coefficients are derived from testing out the game given a fixed depth. The max depth I used to cut off the search is also derived from testing out the game to guarantee the move generated by AI doesn’t take long but can still give a somewhat optimal move. An explanation of how I constructed the heuristic will be detailed below.

Another additional heuristic to limit the search space is to only generate successor nodes (positions) that are near the pre-existing stones on the board. This is implemented by defining a function that generates successor nodes only for positions that are within a certain distance of existing stones. This is reasonable given an isolated square cannot lead to immediate loss/win and add less value to the game. I also used the heuristic to create a random computer moves generator. The constraints make the moves less random since it limits the positions the stone lands on and makes it slightly more effective than a pure random moves generator.

c. Explanation of heuristic function

Given S(i,j) denotes a sequence of length i (where i is an integer ranging from 2 to 5), with j being the number of openings (ranging from 0 to 2). If S(i,j) belongs to the current player (max player), its score is given by weight_i * j, where weight_i is a coefficient (which value is derived from tests) that depends on the length of the sequence i. On the other hand, if S(i,j) belongs to the opponent, its score is -weight_i * j. **The heuristic score is obtained through summing up the scores of all sequences**. The longer the sequence (i) is, the larger the coefficient is. This is because we want the game board to prioritize a long sequence of our stones as it leads to a better chance of winning and is also heavily penalized when the opponent is forming a long sequence to prevent us from losing. The weight (or coefficients) is determined through constant tests. The number of openings (j) is also an important variable, as a sequence with more openings has a higher chance of extending. If a sequence is closed (no opening), it cannot develop into a sequence of five. Therefore, moves that would lead to a sequence with the same length but have more empty cells (openings) around it would be more prioritized.

III. Testing

I will conduct several tests that would confirm that the AI Gomoku Player is operating rationally and have relevant results. Further evidence and details will be listed in Appendix, along with snapshots of the game.

1) Human versus AI: The human player (me) is the white player, and the AI player is the black player. I tested three games, and the AI beat me in all games as I may have made mistakes and performed suboptimal moves. This result is expected as AI is behaving (nearly) optimally, so it should be able to win. We can see when I'm nearing 3 or 4 sequences, the AI player always manages to make defensive moves to block me while also making offensive moves to win. The demo video and screenshots of the game are illustrated in Appendix A.

2) AI vs Random Moves Generator (with constraints): With the AI playing against a Non-AI agent, I tried to test 100 games and plot graphs. The graph shows that the AI agent wins all 100 games, meaning that the moves the AI player made are more informed. However, I observed the AI agent made a lot of suboptimal decisions which prevented it from making a much earlier win. This made sense as the minimax search algorithm assumes that the opponent is playing optimally, which isn't true for the non-AI agent. The bar graph (illustrating the result of the testing of 100 games) and screenshots of the game are illustrated in Appendix B.

3) AI vs AI: With AI vs AI, the result shows a tie (it took almost an hour for the game to finish fully). We can see that there are a lot of sequences of three and four in the game made by both players but were never completed to five. This means the AI manages to balance defensive moves and offensive moves. As both players presumably are playing optimally, it made sense that the game resulted in a tie.

To conclude, various test cases show that the AI agent made an informed decision before taking action and could compete to win the game against humans. Their decision is certainly more informed than the Non-AI agent's as the test shows it won against them in 100 rounds. However, it's worth noticing that the AI assumes that the opponent is making optimal decisions to choose their next move (which isn't always true for humans and random moves generator). This can be a potential explanation for why at the beginning of the game the AI was making suboptimal moves against the non-AI agent (but not against humans). However, when playing against a human who is trying to win the game or another AI agent, the moves are always (nearly) optimal. Furthermore, the depth level is quite small, therefore, sometimes making suboptimal decisions are expected.

IV. AI Use

I only used Chat-GPT to initially understand more about how the game works. However, I read and used other sources to write my paper.

References

Codeofcarson. (n.d.). *Checkers/board.py at master · Codeofcarson/Checkers*. GitHub. Retrieved April 20, 2023, from https://github.com/codeofcarson/Checkers/blob/master/board.py

Luzgabriel. (n.d.). *Luzgabriel/Pymoku: Gomoku Terminal based game written in Python*. GitHub. Retrieved April 20, 2023, from https://github.com/luzgabriel/pymoku

Wikimedia Foundation. (2023, March 19). *Gomoku*. Wikipedia. Retrieved April 20, 2023, from https://en.wikipedia.org/wiki/Gomoku

Russell, S. J., Norvig, P., & Davis, E. (2022). *Artificial Intelligence: A modern approach*. Pearson Educación.

APPENDIX A: TESTING HUMAN VS AI

Simply go to the file directory and run through the terminal command:

python3 -m venv venv

source venv/bin/activate

pip3 install -r requirements.txt

python3 gomoku_game.py

```
============================================================
ooooo                  GOMOKU GAME                    oooooo
============================================================
1. Human vs AI
2. Computer Random vs AI
3. Human vs Computer Random
4. AI vs AI
Enter your choice: 
```

Demo game link: https://www.loom.com/share/2a157d98764d44a29d56f30a8d0c9ccf

1. Round 1: AI wins

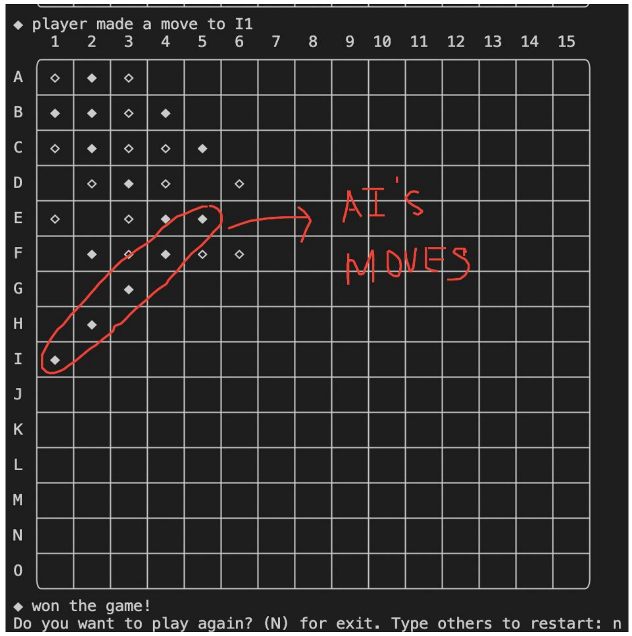

Figure 1: Screenshot showing the result of the first round, where the AI won against the human player. The sequence of five was made diagonally from I1 to E5.

2. Round 2: AI wins

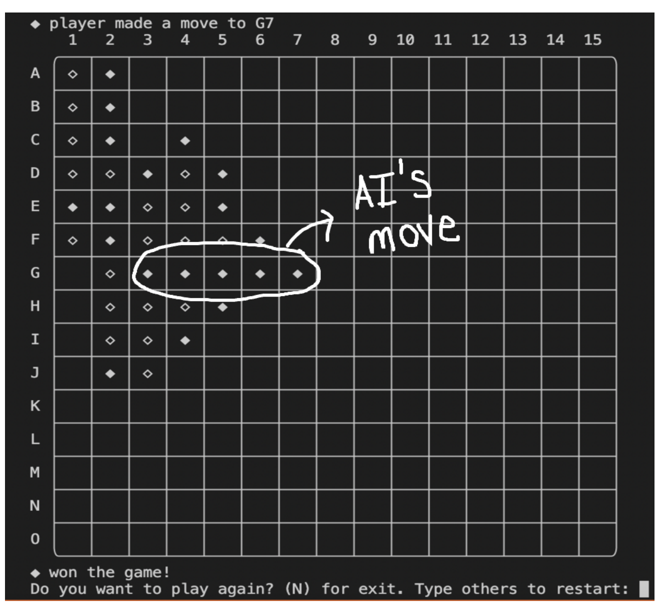

Figure 2: Screenshot showing the second round result, where the AI won against the human player. The sequence of five was made horizontally from G3 to G7.

3. Round 3: AI wins

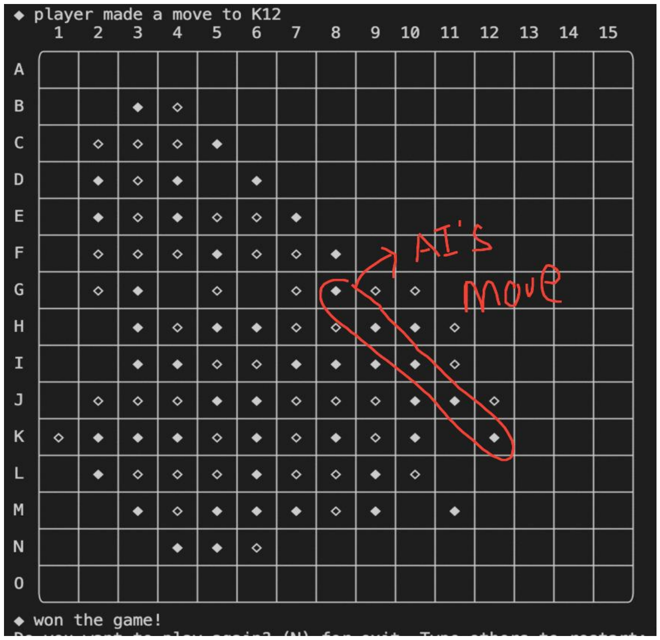

Figure 3: Screenshot showing the result of the third round, where the AI won over the human player again. The sequence of five was made diagonally from G7 to G12.

APPENDIX B: TESTING AI VS NON-AI PLAYER

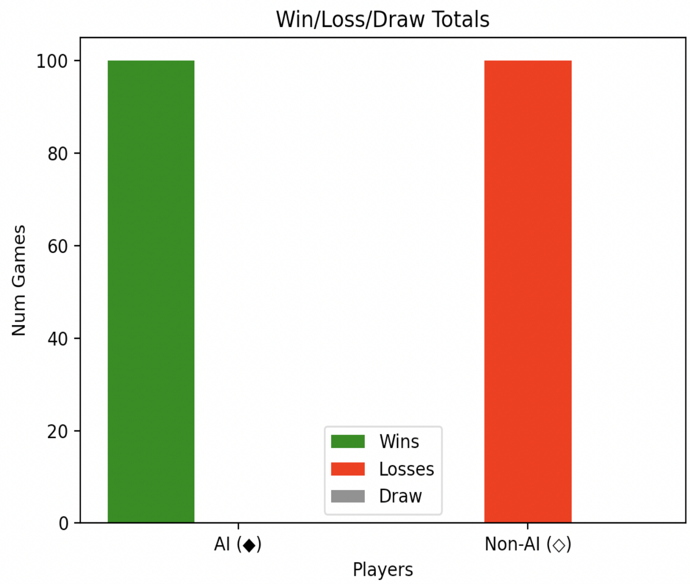

Figure 4: Bar graph showing the number of Wins, Losses, Draw of the AI and Non-AI players who have played 100 rounds of games.

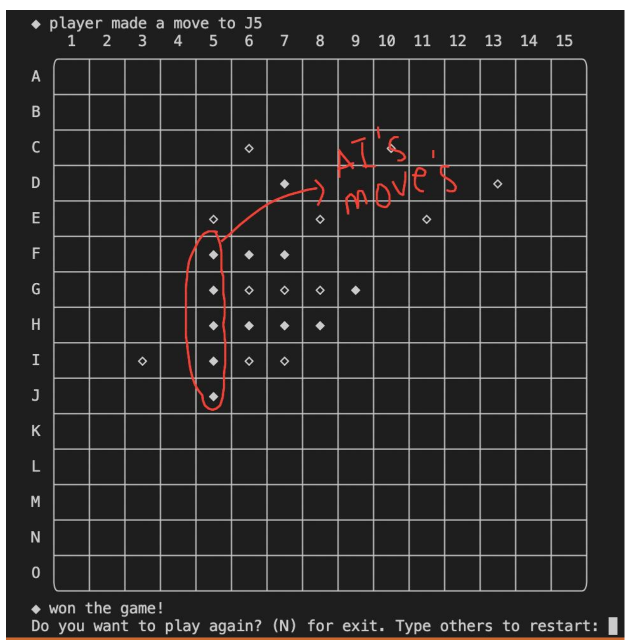

Figure 5: Screenshot showing the result of a game in which the AI won after making a sequence of five vertically from F5 to J5.

APPENDIX C: TESTING AI PLAYER VS AI PLAYER

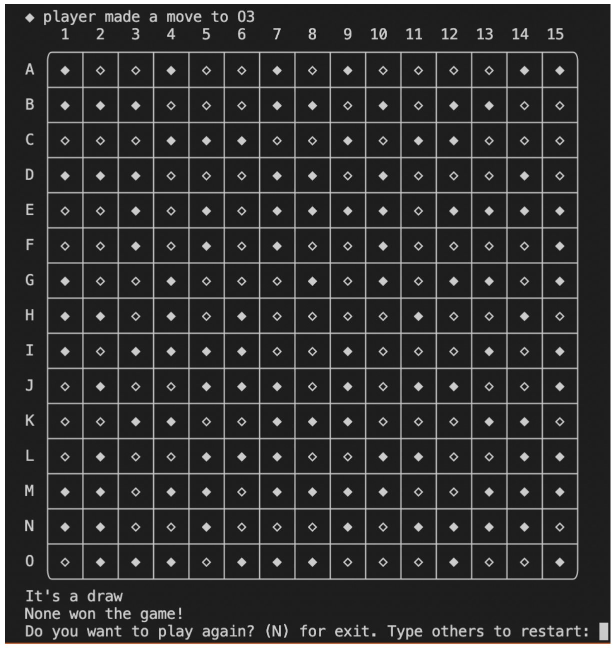

Figure 6: Screenshot showing the game resulting in a draw when they played against each other

APPENDIX D: Proposal Form

Thanks for filling out CS152 Final Project Proposal

Here's what was received.

CS152 Final Project Proposal

Fill out the details below. Please note that it may take up to a week to get a response to your proposal, so it is in your interest to submit it with adequate time prior to the end of week 15.

Your email (**trinhnguyen@uni.minerva.edu**) was recorded when you submitted this form.

Name *

Trinh Nguyen

Problem Definition *

Provide a one-paragraph description of the problem you are trying to solve using an AI method covered in this course, or an extension thereof.

Problem Definition *

Provide a one-paragraph description of the problem you are trying to solve using an AI method covered in this course, or an extension thereof.

Connect 5 (Maybe more like TicTacToe in the interface but instead of 3, we need to have 5 dots)

Targeted LOs *

Indicate the LOs that you plan to apply in your project below. Note that all projects are expected to apply #aicoding in addition to one or more content LOs, unless there are compelling reasons not to (need to discuss with Prof. beforehand). Also, to apply #aiconcepts well, your project would need some investigation into the theoretical/philosophical basis of AI that goes beyond what we covered in terms of rationality and agent programs.

- [x] #aicoding
- [x] #search
- [x] #ailogic
- [ ] #robotics
- [ ] #aiconcepts

Proposed Solution & Deliverables *

Write up to one paragraph describing how you plan to approach your problem. In particular, list the AI algorithm or solution method that you plan on applying in order to solve your problem, and the deliverable you will submit.

Im planning to make connect5 (just run on terminal) ai and use minimax algorithm with alpha-beta pruning to do so.

APPENDIX E: Code

I. gomoku_game.py

```
from player import Player, SelfPlayer, ComputerPlayAI, ComputerPlayerEasy
```

```
import numpy as np
#code logic inspired by https://github.com/luzgabriel/pymoku/blob/master/pymokugame/pymokugame.py
# printing board function slightly inspired by game of checker https://github.com/codeofcarson/Checkers/blob/master/board.py
class GomokuGame():
    """
    Class Gomoku Game: A class to create structure for Gomoku Game with four modes Human vs AI, Humans vs Non-AI agent
    ,Non-AI vs AI (for testing purpose) and AI vs AI (for testing purpose). To start the game,
    ,we method play_game

    --------------------------
    Attribute:
    - width (int): Default to 15. The width of the Gomoku board
    - height (int): Default to 15. The height of the Gomoku board
    - max_depth(int): Default to 2. The cut off depth of the tree (works well with the weight given in the heuristic functions)
    """
    BLACK = "◆"
    WHITE = "◇"
    EMPTY = ' '
    def __init__(self, width=15, height=15,max_depth=2):
        self.width = width
        self.height = height
        #set the initial board state of the game to be all empty
        self.board_state = [[self.EMPTY for j in range(self.width)] for i in range(self.height)]
        self.game_over = False
        self.winner = None
        self.maxdepth = max_depth
        #self.counter = 0
        self.all_moves =[]  #all moves that have been generated by both opponents
        self.all_moves_comp = [] #all moves that have been generated by the AI Agent


    def check_valid_move(self,row,column):
        """
        This function checks whether the move is valid given the current board of the state
        --------------------------
        Input:
            - row(int): the row number that the position in
            - column(int): the column number that the position in
        Output:
            - bool: return True if the position is valid else False
        """
        #check if the position is within the board game size
```

```
        if row >= 0 and row <= self.height -1:
            if column >= 0 and column <= self.width- 1:
                #check to see whether the position is occupied
                if self.board_state[row][column] == self.EMPTY:
                    return True
        else:
            return False


    def available_bounded_pos(self,bound=5):

        """
        This function returns a list of empty positions within a region of bounded box size
        from pre-existing stones:
        ---------------------------
        Input:
            - bound(int): default to five. Width/Height of the bounded box size.
        Output:
            - available_pos(list): return a list of available poisitons that the player can move to
        """
        available_pos = []
        for pos in self.all_moves:
            #get the row, column number
            pos_x = pos[0]
            pos_y = pos[1]
            for i in range(0,bound):
                for j in range(0,bound):
                    #get the row, column of position within the bound of a preexisting stone
                    new_pos_x = pos_x + (i - bound//2)
                    new_pos_y = pos_y + (j - bound//2)
                    #making the sure the position is valid and not repetitive
                    if (new_pos_x >= 0) and (new_pos_y >= 0) and self.check_valid_move(new_pos_x,new_pos_y):
                        if ([new_pos_x, new_pos_y] not in available_pos) and (self.board_state[new_pos_x][new_pos_y] ==
self.EMPTY):
                            available_pos.append([new_pos_x,new_pos_y])
        return available_pos

    def empty_positions(self):
        """
        This function checks whether the position is empty
        ---------------------------
        Input:
```

```
        - row(int): the row number that the position in
        - column(int): the column number that the position in
    Output:
        - bool: return True if the position is empty else False
    """
    for row in self.board_state:
        for pos in row:
            if pos == self.EMPTY:
                return True
    return False


def mark_position(self,position,player,do_print:bool):
    """
    This function mark the position of the position on the board
    --------------------------
    Input:
        - position(list) : the position of the move in row,column
        - player(str)    : the symbol of the player
        - do_print(bool) : print out string showing the player's moves

    Output:
        - bool: return True if the position is marked else return False
    """
    try:
        row = position[0]
        column = position[1]
        #mark poisiton on the board
        if self.check_valid_move(row,column):
            self.board_state[row][column] = player
            if do_print:
                print(f'{player} player made a move to {chr(row+65)}{column+1}')
            return True
        else:
            if do_print:
                print("Board game position is invalid try again")
            return False
    except:
        print("Board game position is invalid try again")
        return False
```

```
def get_row(self, position):
    """
    This function returns the array which is a row (horizontal line) of where the position is at
    --------------------------
    Input:
        - position(list) : the position of the move is at
    Output:
        - list: the array which is a row (horizontal line) of where the position is at
    """
    state = np.array(self.board_state)
    return state[position[0],:]

def get_column(self,position):
    """
    This function returns the array which is a row (horizontal line) of where the position is at
    --------------------------
    Input:
        - position(list) : the position of the move is at
    Output:
        - list: the array which is a column (vertical line) of where the position is at
    """
    state =  np.array(self.board_state)
    return state[:,position[1]]

def horizontal_vertical_sequences(self):
    #instantiate empty list of sequences
    sequences =[]

    #keep track the visited row_column that we have parse the sequences
    visited_pos =[]
    #for every current stone we have
    for pos in self.all_moves:

        #check if we have already got the sequences of column containing that pos
        #if not, parse all sequences in that column and append to list
        if pos[1] not in visited_pos:
            column = self.get_column(pos)
            sequence = self.get_sequences_in_array(column)
            visited_pos.append(pos[1])
            if len(sequence) > 0:
                sequences.extend(sequence)
```

```
        #check if we have already got the sequences of row containing that pos
        #if not, parse all sequences in that row  and append to list
        if pos[0] + 15 not in visited_pos:
            row = self.get_row(pos)
            sequence = self.get_sequences_in_array(row)
            visited_pos.append(pos[0]+15)
            if len(sequence) > 0:
                sequences.extend(sequence)
    return sequences


def get_sequences_in_array(self,array):
    """
    Get a list of sequences in the given array.
    --------------------------------

    Input:
        - array(list): A list representing the array we want to get the sequence of.

    Output:
        list: A list of sequences. Each element of the list is a list containing information about a sequence,
        including the player symbol, the number of open ends, and the length of the sequence.
    """

    sequences = []
    #instantiate the sequence of the current array
    temp_seq = []
    temp_open = 0

    #check if the sequence is open, if the sequence is blocked both sides 0
    #otherwise it's either 1 or 2 (represents number of open sides )
    for idx, val in enumerate(array):
        #skip the first element as there is no prev item we can compare
        #to see if its the sequence >1
        if idx == 0:
            continue
        else:
            prev_val = array[idx-1]
            # if current value is occupied by either player
            if val != self.EMPTY:
                if prev_val != val:
                    #if previous value is unoccupied, the opening of the sequence is at least 1
```

```
                if prev_val == self.EMPTY:
                    temp_open = 1
                    temp_seq.append(val)

                #if the previous val is occupied by the oppenent
                else:
                    #if the length prev sequence is larger than one, append to overall sequences
                    if len(temp_seq) > 1:
                        sequences.append([temp_seq[0],temp_open, len(temp_seq)])

                    #restart sequence and opening
                    temp_seq = []
                    temp_open = 0

            # if the previous value is the same as current val
            else:
                #make sure to append the starting item  of the list to seq
                if(len(temp_seq)) < 1:
                    temp_seq.append(val)
                #append the current val to seq if its the same as prev
                temp_seq.append(val)

        #check the end of a sequence
        #if the previous val is not equal to current value
        #the prev value may be the end of the sequence
        elif prev_val != val:
            if len(temp_seq) > 1:
                #append previous sequence to list
                sequences.append([temp_seq[0], temp_open+1, len(temp_seq)])
            #close sequence and restart variable
            temp_seq = []
            temp_open = 0

    #make sure we append the last sequence of the array to the list
    if len(temp_seq) > 1:
        sequences.append([temp_seq[0], temp_open, len(temp_seq)])
    sequences = [sequence for sequence in sequences if sequence[0] != self.EMPTY]
    return sequences


def diagonal_line(self):
    """
```

```
        Get a list of diagonal line in a board
        -------------------------------

        Input:
            None

        Output:
            list: A list of all diagonal lines in the board (represented as list).
        """
        sequences = []
        matrix = np.array(self.board_state)
        diagonal_lines = []

        # append diagonal lines that span from top-left to bottom-right
        for i in range(-matrix.shape[0] + 1, matrix.shape[1]):
            diagonal_lines.append(matrix[::-1,:].diagonal(i))

        # append diagonal lines that span from top-right to bottom-left
        for i in range(matrix.shape[1] - 1, -matrix.shape[0], -1):
            diagonal_lines.append(matrix.diagonal(i))

        # get all diagonal sequences of all diagonal lines
        for diagonal in diagonal_lines:
            sequences += self.get_sequences_in_array(list(diagonal))

        #make sure that only get sequences that are from bklack or white player
        sequences = [sequence for sequence in sequences if sequence[0] != self.EMPTY]
        return sequences

    def get_total_sequences(self):
        """
        Get all the total sequences in the board
        -------------------------------

        Input:
            None

        Output:
            list: A list of all diagonal lines in the board (represented as list).
        """
        total_sequences = []
        vertical_horizontal_sequences = self.horizontal_vertical_sequences()
```

```
        #print(f"vertical_horizontal{vertical_horizontal_sequences}")
        diagonal_sequences = self.diagonal_line()
        #print(f"Diagnonal{diagonal_sequences}")
        total_sequences.extend(vertical_horizontal_sequences)
        total_sequences.extend(diagonal_sequences)
        #print(total_sequences)
        return total_sequences

    def win_game(self):
        """
        check to see if someone has won a game given the state
        return the winner if so
        --------------------------------

        Input:
            None

        Output:
            str: The winner of the game (if there is)
        """

        total_sequences = self.get_total_sequences()
        for sequence in total_sequences:
            if len(sequence) > 0:
                #check if there is any sequence larger than 5
                if sequence[2] >= 5:
                    self.winner = sequence[0]
                    return sequence[0]
        return None

    def get_score(self,length):
        """
        getting the weight of the sequence given a length
        --------------------------------

        Input:
            length(int): length of the sequence

        Output:
            int: the weight of the sequence given a length

        """
```

```
    if length == 2:
        return 1
    elif length == 3:
        return 800
    elif length == 4:
        return 60000
    elif length >= 5:
        return 4000000000

def undo_move(self,position):
    """
    undoing a move
    -------------------------------


    Input:
        pos(list): position of a move we want to undo

    Output:
        None

     """
    self.board_state[position[0]][position[1]] = self.EMPTY

def calculate_heuristic_score(self, player, depth):
    """
    calculate heuristic score of the board configuration
    -------------------------------


    Input:
        - player(str): the current player

    Output:
        - heuristic_score(int): heuristic score of the board configuration

     """
    #get all sequences in a board configuration
    total_sequences = self.get_total_sequences()
    #initialize the heuristic score
    heuristic_score = 0

    #get score and information of each sequence
    for sequence in total_sequences:
```

```
        if len(sequence) > 0:
            #get the player's symbol whose sequence it belongs to
            player_val = sequence[0]
            #get number of opening
            opening = sequence[1]
            #get the length of the sequence
            length_seq = sequence[2]

            if length_seq in range(2, 5):
                #get the score of each sequence: each sequence score  (length 2 to 4): score(given length) * number of opening
                seq_score = self.get_score(length_seq) * opening
                #add the seq score to the total heuristic score
                #if the sequence belongs to the player
                #penalize if it belongs to the opponent
                if player_val == player:
                    heuristic_score += seq_score
                else:
                    heuristic_score -= seq_score

            elif length_seq >= 5:
                #get seq score if it is equal or larger than 5
                #as the num openings doesn't matter (since you would lose/win anyway): sequence score = score(5)
                #create a new function
                seq_score = self.get_score(5)
                if player_val == player:
                    heuristic_score += seq_score
                else:
                    heuristic_score -= seq_score
    return heuristic_score
    #return (heuristic_score*225)/depth


def minimax_with_pruning(self, player, alpha=float('-inf'), beta=float('inf'), count_depth=0, positions_list=None):
    """
    implement the minimax algorithm with alpha-beta pruning to determine the optimal move.
    ------------------------
    input:
        player (str): the current player
        alpha (float): the alpha value for alpha-beta pruning. initialized as negative infinity.
        beta (float): the beta value for alpha-beta pruning, initialized as positive infinity.
        count_depth (int): the current depth of the recursive search, default as 0.
        positions_list (list): The list of all previous moves, default = None
```

```
output:
    list: return a list containing the optimal score and move.
"""

# get the list of possible next positions
next_positions = self.available_bounded_pos()
if positions_list is None:
    positions_list = self.all_moves.copy()

# initialize the optimal move and score
opt_move = [-1,-1]

# check if the game is over or if the maximum depth has been reached
if not self.empty_positions() or count_depth >= self.maxdepth:
    # if the game is over or the maximum depth has been reached, calculate the heuristic score for the current board state
    heuristic_score = self.calculate_heuristic_score(player, count_depth)
    # return the heuristic score and the optimal move
    return (heuristic_score, opt_move)

else:
    # loop through the possible next positions
    for position in next_positions:
        # make the move and update the position list
        self.mark_position(position, player, do_print=False)
        positions_list.append(position)

        # get the next player and calculate the heuristic score recursively
        next_player = self.WHITE if player == self.BLACK else self.BLACK
        # call the minimax_with_pruning function recursively with the updated position and player information
        # the count_depth parameter is incremented by 1 to indicate that we are going one level deeper in the search tree
        # the positions_list is copied to ensure that each recursive call operates on its own independent list of previous moves
        heuristic_score = self.minimax_with_pruning(next_player, alpha, beta, count_depth+1, positions_list.copy())[0]

        # undo the move and update the position list
        self.undo_move(position)
        positions_list.pop()

        # update the alpha-beta values and optimal move
        # if the current player is the maximizer, update the alpha value if the heuristic score > larger
        #as we find better move
        if player ==  self.WHITE:
```

```
            if heuristic_score > alpha:
                alpha = heuristic_score
                opt_move = position

        # if the current player is the minimizer, update the beta value if the heuristic score < lower
        #as the minimizer find better move
        else:
            if heuristic_score < beta:
                beta = heuristic_score
                opt_move = position


        # if the alpha value is greater than or equal to the beta value, pruning can be done
        if alpha >= beta:
            return [alpha if player == self.WHITE else beta, opt_move]

    # return the appropriate score and optimal move depending on whether the current player is the maximizer or minimizer
    return [alpha if player == self.WHITE else beta, opt_move]



def get_opt_move(self):
    """
    get optimal move for a player
    --------------------------------
    Input:
        None

    Output:
        return optimal move

    """
    _, opt_move = self.minimax_with_pruning(self.WHITE, alpha=float('-inf'), beta=float('inf'), count_depth=0)
    return opt_move

def print_board(self):
    """
    print  out the board game
    --------------------------------
    Input:
        None
```

```
    Output:
        str: Print out the state if the board game

    """
    board_line = []
    board_line.append("    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 ")
    board_line.append(f'  ╭{"───┬" * (self.width-1)}───╮ ')

    for num, row in enumerate(self.board_state[:-1]):
        board_line.append(f'{chr(num+65)} │ {" │ ".join(row)} │')
        board_line.append(f'  ├{"───┼" * (self.width-1)}───┤')

    board_line.append(f'{chr(self.height+64)} │ {" │ ".join(self.board_state[-1])} │')

    board_line.append(f'  ╰{"───┴" * (self.width-1)}───╯ ')
    print('\n'.join(board_line))



def play_game(self, black_player, white_player, board_print=True):
    """
    Set the game to start
    --------------------------------
    Input:
        - black_player(str): symbol of first player
        - black_player(str): symbol of second player
        - board_print(bool): whether to print out the board or not
    Output:
        - A game that we can play
    """

    while True:
        turn = self.WHITE
        print(" ======================================================================")
        print(" ooooo                  GOMOKU  GAME                     oooooo ")
        print(" ======================================================================")
        if board_print:
            self.print_board()
        black_turn = False
        self.winner = None  # reset the winner for each game
        while self.winner != self.BLACK and self.winner != self.WHITE:
            if not self.empty_positions():
```

```
                print("It's a draw")
                break
            if black_turn:
                filled_pos = black_player.get_move(self)
                if board_print:
                    self.print_board()
                black_turn = False
            else:
                filled_pos = white_player.get_move(self)
                if board_print:
                    self.print_board()
                black_turn = True
            self.winner = self.win_game()
        print(f"{self.winner} won the game!")
        play_again = input("Do you want to play again? (N) for exit. Type others to restart: ").lower()
        if play_again == "n":
            break
        else:
            self.__init__(15,15,2)
            self.play_game
              # reset the game board for the new game



if __name__ == '__main__':
    # prompt the user to play with AI or AI vs BOt
    print(" ==================================================================")
    print(" ooooo                GOMOKU GAME                 oooooo ")
    print(" ==================================================================")
    print("1. Human vs AI")
    print("2. Computer Random vs AI")
    print("3. Human vs Computer Random")
    print("4. AI vs AI ")

    choice = input("Enter your choice: ")
    try:
        #setting up the mode for human vs ai
        if int(choice) == 1:
            black_player = ComputerPlayAI("◆")
            white_player = SelfPlayer('◇')
            game = GomokuGame()
            game.play_game(black_player,white_player,board_print=True)
```

```
    #setting up the mode for non-AI vs AI
    elif int(choice) == 2:
        black_player = ComputerPlayAI("◆")
        white_player = ComputerPlayerEasy('◇')
        game = GomokuGame()
        game.play_game(black_player,white_player,board_print=True)

    #setting up the mode for human vs non-AI
    elif int(choice) == 3:
        black_player = SelfPlayer('◆')
        white_player = ComputerPlayerEasy('◇')
        game = GomokuGame()
        game.play_game(black_player,white_player,board_print=True)

    #setting up the mode for AI vs AI
    elif int(choice) == 4:
        black_player = ComputerPlayAI('◆')
        white_player = ComputerPlayAI('◇')
        game = GomokuGame()
        game.play_game(black_player,white_player,board_print=True)
    else:
        raise ValueError("Invalid choice")
except ValueError as e:
    print(f"Error: {e}")
```

2. player.py

```
import random
class Player:
  """
  A parent class representing a player in the Gomoku game.
  ---------------------
```

```
  Attribute:
    - player(str): The player's symbol

  """

  def __init__(self,player):
      self.player = player

  def get_move(self, game):
      pass

class ComputerPlayerEasy(Player):
  """
  A child class of the Player class representing a random moves generator
  --------------------
  Attribute:
    - player(str): The player's symbol

  """
  def __init__(self, player):
      super().__init__(player)



  def get_move(self, game):
      if len(game.available_bounded_pos()) == 0:
          move = [6,6]
      else:
          move = random.choice(game.available_bounded_pos())
      game.mark_position(move, player=self.player, do_print=True)
      game.all_moves_comp.append(move)
      game.all_moves.append(move)
      return move

class SelfPlayer(Player):
  """
  A child class of the Player class representing a human player.
  --------------------
  Attribute:
    - player(str): The player's symbol

  """
```

```
  def __init__(self, player):
    super().__init__(player)

  def get_move(self, game):
    input_val = None
    valid_move = False
    while not valid_move:
      player_input = input(f"It's {self.player}'s turn. Enter the game position in row/column form (eg: A4):")
      player_input = player_input.strip()
      input_val = self.parse_input(player_input)
      valid_move = game.mark_position(input_val, player=self.player, do_print=True)
    game.all_moves.append(input_val)
    return input_val

  def parse_input(self,player_input):
    #parsing input of users to get correct index
    try:
      row = player_input[0].upper()
      row = ord(row.upper()) - 65
      column = int(player_input[1:])-1
      return [int(row), column]
    except:
      return False, False


class ComputerPlayAI(Player):
  """
  A child class of the Player class representing a computer AI agent.
  --------------------
  Attribute:
    - player(str): The player's symbol

  """
  def __init__(self, player):
    super().__init__(player)
  def get_move(self, game):
    move = game.get_opt_move()
    game.mark_position(move, player=self.player, do_print=True)
    game.all_moves_comp.append(move)
    game.all_moves.append(move)
    return move
```

3. graph.py

```
if __name__ == '__main__':
  #set up game
  black_player = ComputerPlayAI("◆")
  white_player = ComputerPlayerEasy('◇')
  game = GomokuGame(15, 15, 2)
  #https://realpython.com/python-matplotlib-guide/#bar-charts
  #track results showing win/lose/draw
  resultss =[]
  results = {'Wins': [0, 0], 'Loss': [0, 0], 'Draw': [0, 0]}
  for i in range(100):
    result = game.play_game(black_player, white_player, board_print=False)
    if result == '◆':
      results['Wins'][0] += 1
      results['Loss'][1] += 1
    elif result == '◇':
      results['Wins'][1] += 1
      results['Loss'][0] += 1
    else:
      results['Draw'][0] += 1
      results['Draw'][1] += 1
    resultss.append(result)
  print(results)



  x = range(2)
  fig, ax = plt.subplots()
  bar_width = 0.3

  #plot bars showing win/lose/draw
  ax.bar(x, results['Wins'], width=bar_width, color='green', label='Wins')
  ax.bar([i + bar_width for i in x], results['Loss'], width=bar_width, color='red', label='Losses')
  ax.bar([i + 2*bar_width for i in x], results['Draw'], width=bar_width, color='gray', label='Draw')

  # add axis labels and legend
  ax.set_xlabel('Players')
  ax.set_ylabel('Num Games')
  ax.set_title('Win/Loss/Draw Totals')
  ax.set_xticks([i + bar_width for i in x])
  ax.set_xticklabels(['AI (◆)', 'Non-AI (◇)'])
```

```
ax.legend()

# show plots
plt.show()
```