T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE4082 - Artificial Intelligence

Homework #2 Report

Group Members

150119692 - Bedirhan SARIHAN

170517033 - Yasin Alper BİNGÜL

4-connect is a two-player board game in which the players take turns placing pieces on a grid. The goal of the game is to be the first player to create a sequence of four of your own pieces in a row, either horizontally, vertically, or diagonally. Players can only place their pieces on an empty space on the grid, and the game ends when one player creates a 4-in-a-row sequence or when the grid is filled and no sequence has been created.

The game can be played on a variety of board sizes, but a common choice is a 8x7 grid. Players can use any type of game piece, such as coins, pieces of paper, or game tokens.

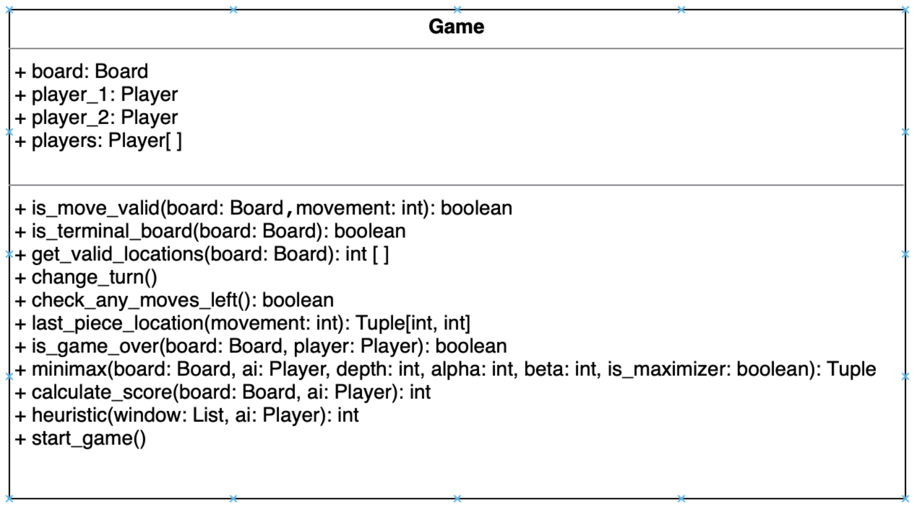
4-connect is a simple but challenging game that requires strategic thinking and careful planning. It can be played by people of all ages and is a good activity for developing problem-solving skills.

We solve the 4-connect game with using some techniques down below:

1. **Minimax algorithm**: This is a common technique used in two-player games such as Connect Four. The basic idea is to have the AI player assume that the opponent will always make the move that is most advantageous to them, and to choose the move that minimizes the maximum loss (hence the name minimax).
2. **Alpha-beta pruning:** This is a optimization technique that can be used to improve the efficiency of the minimax algorithm. It works by pruning (eliminating) certain branches of the search tree, based on the fact that one player's gain is always the other player's loss.
3. **Evaluation function:** This is a function that is used to assign a numerical value to each possible game state. The AI can then use this value to evaluate different moves and choose the one that leads to the most favorable game state. An evaluation function is a function that is used to assign a numerical value to each possible game state in a board game such as Connect Four. This value can be used to evaluate the relative "goodness" of different game states, with higher values indicating more favorable states for the player and lower values indicating less favorable states. There are many different ways to design an evaluation function, and the specific approach will depend on the characteristics of the game and the goals of the AI. So we designed 3 different approach to solve this problem. These are defined as *easy level, medium level* and *hard level* approaches. We will delve into a more comprehensive examination of these algorithms in a later section of this report.

Game Software Design

- ### Game Class



```
Game
+ board: Board
+ player_1: Player
+ player_2: Player
+ players: Player[ ]

+ is_move_valid(board: Board, movement: int): boolean
+ is_terminal_board(board: Board): boolean
+ get_valid_locations(board: Board): int [ ]
+ change_turn()
+ check_any_moves_left(): boolean
+ last_piece_location(movement: int): Tuple[int, int]
+ is_game_over(board: Board, player: Player): boolean
+ minimax(board: Board, ai: Player, depth: int, alpha: int, beta: int, is_maximizer: boolean): Tuple
+ calculate_score(board: Board, ai: Player): int
+ heuristic(window: List, ai: Player): int
+ start_game()
```

The Game class class initializes with two player instances, a "**player_1**" and a "**player_2**", and creates an instance of a "**Board**" class.

The class contains several methods that are used to check for game conditions, such as "**is_move_valid**", "**is_terminal_board**", "**is_game_over**", and "**get_valid_locations**". It also contains methods that are used to change turns and check if there are any moves left.

The **is_game_over** method checks for a win condition with **calculate_score** function by checking the horizontal, vertical, and diagonal lines of the board and returns true if any of the lines have four consecutive pieces of the same color.

The **minimax algorithm** which is a decision-making algorithm used in artificial intelligence, game theory, and operations research. The algorithm is applied to the Connect Four game to determine the best move for the AI player. The minimax function takes several arguments: the current board, the AI player, the current search depth, alpha and beta values for the alpha-beta pruning, and a Boolean value indicating whether the current search is a maximizing or minimizing search.

The function starts by getting all the possible moves from the current board using the get_valid_locations method. It then checks if the current search has reached the maximum depth or if the game is in a terminal state. If either of these conditions is met, the function returns a score for the current board state. If the game is over and the AI player wins, the score is set to 10000. If the human player wins, the score is set to -10000. If there is a tie, the score is set to 0. If the search depth is 0, the function uses the calculate_score method to determine the score for the current board.

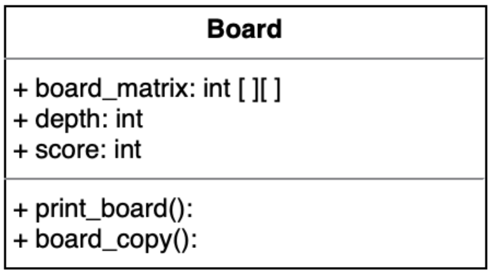
If the current search is a maximizing search, the function loops through all the possible moves, making a copy of the board for each move, applying the move to the copy, and calling the minimax function recursively with the new board, player, and updated search depth. The function keeps track of the best move and the highest score, updating them if a new move results in a higher score. It also uses alpha-beta pruning to stop searching a branch of the tree if the score is higher than the beta value.

If the current search is a minimizing search, the function does the same process as the maximizing search, but it keeps track of the best move and the lowest score, updating them if a new move results in a lower score. It also uses alpha-beta pruning to stop searching a branch of the tree if the score is lower than the alpha value. The function returns the best move and the corresponding score as a tuple. This code should be functional for returning best move for AI player in Connect Four Game.

The **calculate_Score** algorithm which calculates a score for a given game board and player. The function starts with a score of 0, and then checks for lines of 4 in a vertical, horizontal, and diagonal direction on the game board. For each line, it calls another function called "**heuristic**" and passes in the current line and the player. The heuristic function calculates a score for that line and returns it. The function then compares the returned score from the heuristic function to the current score and if it is greater, it updates the score. Finally, the function returns the final score.

The **start_game** function which starts a game of Connect Four. The function uses a while loop that continues until the game is over. The function first checks if it is the turn of player 1, if it is, it will either ask for input from a human player to determine the next move, or it will use the minimax algorithm to determine the next move for an AI player. It will check if the move is valid, if it is, it will make the move and check if the game is over, if it is, it will print the winner and the board, if not it will change the turn and print the board. If the move is not valid it will inform the user that the column is not empty. The same process is repeated for player 2.

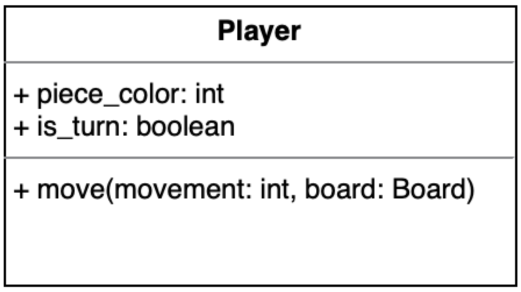
- **Board Class**



The constructor of Board class initializes a new instance of the class. If a board is not passed in, it creates an 8x8 board with all elements set to 0. If a board is passed in, it sets the board matrix to that board. It also initializes depth and score to 0.

board_copy function returns a copy of the current board. It creates a new 2D list by iterating over the rows and elements of the current board and appending them to the new list. It then creates a new Board object with this new board and returns it.

- **Player Class**



In its constructor , it initializes a new instance of the class. It takes in two parameters, piece_color and is_turn. piece_color is an integer representing the color of the pieces that the player is using (1 or 2), and is_turn is a boolean value indicating whether it is the player's turn or not.

Also in the **move** function, it takes in 3 parameters: movement, board, *args, and **kwargs. movement is an integer representing the column where the player wants to make the move, board is an instance of the Board class, and *args and **kwargs are additional arguments that the method can take in. It uses a nested for loop to iterate over the rows and elements of the board_matrix attribute of the board object. When it finds the column index that matches the movement passed in, it checks if the element is 1 or 2 or all the elements of the corresponding column are zero. If it finds an empty space, it sets the element of the topmost empty row in that column to the player's piece color, which is stored in the piece_color attribute of the player object and return None.

Heuristics

According to heuristic function in the Game class, it defines a variable score initialized to 0, and another variable opp_player which is equal to 2 if the current player's piece color is 1 and 1 if the current player's piece color is 2.

The code then checks the difficulty attribute of the player object, which can be 'easy', 'medium', or 'hard', and performs different actions based on the value.

In the **'easy'** difficulty:

- If the opponent has 3 pieces next to each other, and there is one empty space, the score is set to -1.

In the **'medium'** difficulty:

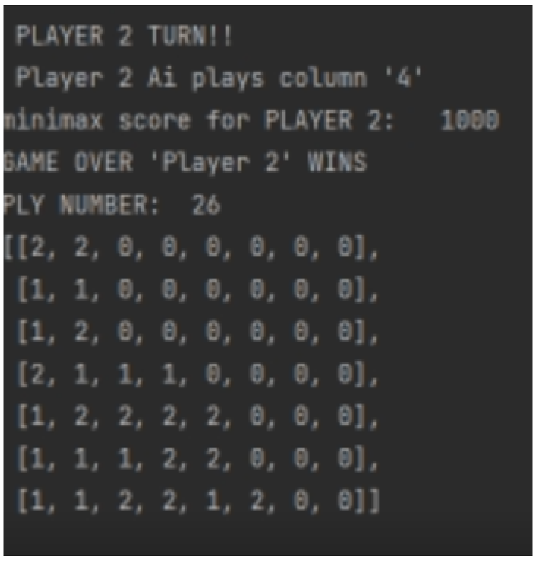
- If the current player has 3 pieces next to each other and one empty space, the score increases by 5.
- If the current player has 2 pieces next to each other and two empty spaces, the score increases by 2.
- If the opponent has 3 pieces next to each other and one empty space, the score is set to -1.

In the **'hard'** difficulty:

- Same as medium difficulty, but also:
- If the opponent has 2 pieces next to each other and 2 empty spaces, the score decreases by 2
- If the opponent has 3 pieces next to each other and 1 empty space, the score is set to -1

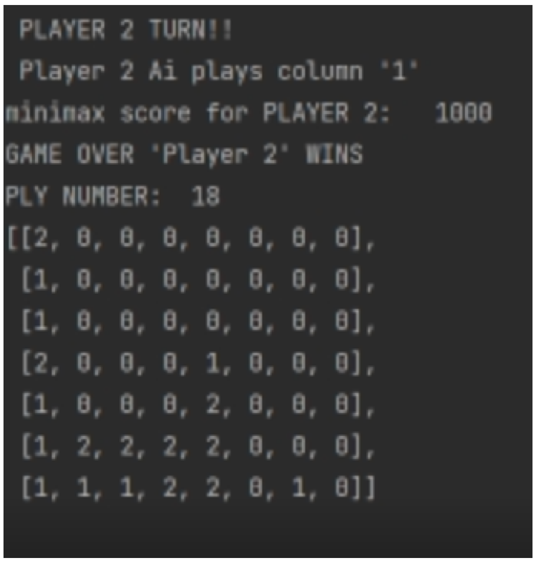
Program Results

When we run the program by selecting the heuristics as **easy** and **medium** with **5 depth**, we get results like down below. As a result, each AI player makes 13 plies in this case. Also the medium one wins.



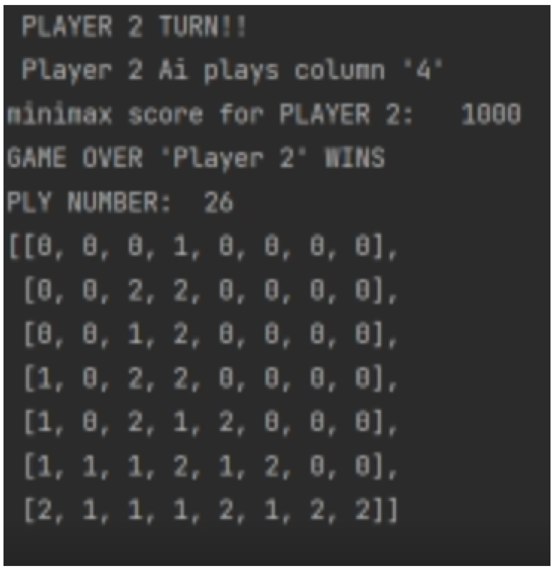
```
 PLAYER 2 TURN!!
 Player 2 Ai plays column '4'
minimax score for PLAYER 2:   1000
GAME OVER 'Player 2' WINS
PLY NUMBER:  26
[[2, 2, 0, 0, 0, 0, 0, 0],
 [1, 1, 0, 0, 0, 0, 0, 0],
 [1, 2, 0, 0, 0, 0, 0, 0],
 [2, 1, 1, 1, 0, 0, 0, 0],
 [1, 2, 2, 2, 2, 0, 0, 0],
 [1, 1, 1, 2, 2, 0, 0, 0],
 [1, 1, 2, 2, 1, 2, 0, 0]]
```

When we run the program by selecting the heuristics as **easy** and **hard** with **5 depth**, we get results like down below. As a result, each AI player makes 9 plies in this case. Also the hard one wins.



```
 PLAYER 2 TURN!!
 Player 2 Ai plays column '1'
minimax score for PLAYER 2:   1000
GAME OVER 'Player 2' WINS
PLY NUMBER:  18
[[2, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 0, 0, 0, 0, 0, 0],
 [2, 0, 0, 0, 1, 0, 0, 0],
 [1, 0, 0, 0, 2, 0, 0, 0],
 [1, 2, 2, 2, 2, 0, 0, 0],
 [1, 1, 1, 2, 2, 0, 1, 0]]
```

When we run the program by selecting the heuristics as **hard** and **medium** with **5 depth**, we get results like down below. As a result, each AI player makes 13 plies in this case. Also the hard one wins.



```
 PLAYER 2 TURN!!
 Player 2 Ai plays column '4'
minimax score for PLAYER 2:   1000
GAME OVER 'Player 2' WINS
PLY NUMBER:  26
[[0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 2, 2, 0, 0, 0, 0],
 [0, 0, 1, 2, 0, 0, 0, 0],
 [1, 0, 2, 2, 0, 0, 0, 0],
 [1, 0, 2, 1, 2, 0, 0, 0],
 [1, 1, 1, 2, 1, 2, 0, 0],
 [2, 1, 1, 1, 2, 1, 2, 2]]
```