Developer document ~TicTacToe on 6 by 6 board~

This code consists of 3 classes.
Algorithmically, the most important part is “3. AI player class”, and I involved a mini - max algorithm to let the AI player make a decision.

1. Game Board class

This class handles the state of the Game Board, constructing, printing, validating the move, and checking the winning state of the movement.
Each method works as follows.

- **InitializeBoard()**: Initialises the board by setting all cells to empty (' ').
- **IsValidMove(int x, int y)**: Checks if a move is valid, ensuring that the selected cell is either empty or has a neighbouring stone.
- **HasNeighbor(int x, int y)**: Helper method to determine if the selected cell has any neighbouring stones.
- **MakeMove(int x, int y, char player)**: Place a player's stone at the specified position if the move is valid.
- **UndoMove(int x, int y)**: Removes a stone from the specified position, reverting the move.
- **CheckWin(char player)**: Checks if the specified player has won by forming a line of 4 consecutive stones either horizontally, vertically, or diagonally.
- **IsFull()**: Checks if the board is full, which would indicate a draw if no player has won.
- **IsEmpty()**: Checks if the board is completely empty.
- **GetAvailableMoves()**: Returns a list of all valid moves (empty cells that can be legally filled).
- **PrintBoard()**: Prints the board with row and column numbers for easier reference.

2. Player class (abstract)

This player class is abstract base class that defines the basic structure of a player in the game. This class is inherited to two subclasses, Human Player class and AI Player class and it only has one method.

- **GetMove(GameBoard board)**: This is an abstract method that must be implemented by derived classes, Human Player class and AI Player class, to return the next move (coordinates) for the player.

3. Human Player class

This “Human Player class” is a subclass of the Player class, which is written above. It handle the movement of the Human Player.

- **GetMove(GameBoard board)**: Asks the human player to enter the row and column for their move and returns the selected coordinates.

4. AI Player class

This class invented the movement of AI players. This is the main “statistical” part of this project. Basic tactics to make a decision is based on a mini - max algorithm, and I also applied alpha-beta pursing to improve the processing speed.
Also, I implemented **“IsCriticalmove”** to improve the defending/blocking action of the Human player’s movement to win**.**

The detail about the methods are as follow

- **GetMove(GameBoard board)**: Implements the Minimax algorithm to evaluate possible moves and select the best one.
- **Minimax(GameBoard board, int depth, bool isMaximizing, int alpha, int beta)**: The core of the Minimax algorithm. It recursively evaluates possible game states to determine the best move. The algorithm uses alpha-beta pruning to eliminate branches of the game tree that are not worth exploring, which improves efficiency.
- **IsCriticalMove(GameBoard board, int x, int y, char opponentSymbol)**: Determines if a move is critical, which means that it could allow the opponent to win on their next turn if not blocked.
- **CheckPotentialLine(GameBoard board, int startX, int startY, int dx, int dy, char opponentSymbol)**: Checks a potential line (row, column, or diagonal) for opponent stones to identify critical moves.
- **EvaluateBoard(GameBoard board)**: Evaluates the board by assessing each line of stones and assigning a score based on the number of consecutive stones for the AI and the opponent.
- **EvaluateLine(GameBoard board, int startX, int startY, int dx, int dy)**: Evaluates a specific line (row, column, or diagonal) and assigns a score based on the AI's and opponent's stones in that line.

And here are some crucial ideas of my algorithm in detail: how AI makes a desicion.

- **Minimax Algorithm**: The AI recursively evaluates all possible future moves up to a certain depth (controlled by maxDepth). It assumes that both players will play optimally:
    - **Maximising Player (AI)**: The AI tries to maximise its score by selecting the move that leads to the best possible outcome for itself.
    - **Minimising Player (Opponent)**: The AI assumes that the opponent will try to minimise the AI's score, so it considers the worst-case scenario in its evaluations.
- **Alpha-Beta Pruning**: To improve efficiency, the AI uses alpha-beta pruning to cut off branches of the game tree that cannot influence the final decision. This reduces the number of nodes the AI needs to evaluate, allowing it to make decisions faster.

- **Critical Moves and Blocking**: The AI prioritises moves that block the opponent from forming a line of 4 stones. If the opponent has 3 consecutive stones, the AI will try to block that line to prevent a loss.
- **Positional Advantage**: The AI also considers the position of the stones on the board. Central positions are valued higher because they offer more opportunities to form lines.

5. Game class

This Game class deals with the overall flow of the game, let the human player and the AI player take turns.

- **Start()**: Starts the game and manages the main game loop. It alternates turns between the human player and the AI, printing the board after each move and checking for a winner. The game loop continues until one player wins or the board is full.
- **Main(string[] args)**: The entry point of the program. It initialises the players (human and AI), creates the game instance, and starts the game.