Youssef Haitham 233579, Moaz Eslam 235960, Mohamed Khaled 235903, Fares Sameh 229306

Go-Moku AI

Group 20

Table of Contents

Contribution Table:

Group Member	ID	Contribution:
Youssef Haitham	233579	All members worked on their assigned parts diligently
Moaz Eslam	235960	
Mohamed Khaled	235903	
Fares Sameh	229306	

Project Overview

In this project, we deployed an intelligent Gomoku player. Our goal was to devise an AI which would play competitively against human players using advanced techniques in Artificial Intelligence, viz. the Minimax algorithm supported by Alpha-Beta pruning and heuristic evaluation functions specifically designed.

Gomoku, or 'Five in a Row,' is played on a 15x15 grid upon which alternately from a reservoir of markers, players make their moves, with the aim to achieve five in a row, either horizontal, vertical, or diagonal. Our AI doesn't merely respond but also look ahead, make rational assessments of positions on the board, ward off threats, and chase winning moves through foresight of strategy.

The project also aims at delivering an interactive and smooth user experience, with an easy yet responsive GUI created using Pygame. Users have the option of dynamically setting board size and resuming games at a snap. Different modes of difficulty were created by switching between two heuristic function types — a simple material-counting model and a more involved strategic evaluation structure.

Finally, this project illustrates the blending of game theory, optimization methods, and heuristic design to construct a fun and a challenging Gomoku-playing AI.

Main Functionality

Other than drawing the board and making the platform where the game will be played. The first thing the AI does is the [checkWinner] function this function moves across the board checks every block if the block is empty it goes to the next block if not then it check if it is in a straight line with other stones of the same type in all directions using the [checkVertical], [checkHorizontal], [checkDiagonal], [checkAntiDiagonal] and if they add up to 5 to find the winner.

After checking for the winner and after the player plays his move the AI gets the position of the click to know where the player played his stone and start from there after the [AI_Player] the AI checks if there a move that would guarantee a victory or loss like if the AI had already placed 4 stones in a row or if the human player had already placed 4 stones in a row it then plays

the best move which in the former is a win or in the latter preventing a win. Then [minimax] is used to determine what the next move will be, while all of this is happening the function [checkWinner] is being run at every recursion of [minimax] and in the function [oneWinMove].

Methodology

Overview:

In order to design a smart Gomoku player, we employed classical game theory algorithms with extensions through AI search optimisations. We employed decision-making through Minimax, optimality through Alpha-Beta pruning, and two stages of heuristic approximations for adaptive response to difficulty levels in the problem. In order to design a smart Gomoku player, we employed classical game theory algorithms with extensions through AI search optimisations. We employed decision-making through Minimax, optimality through Alpha-Beta pruning, and two stages of heuristic approximations for adaptive response to difficulty levels in the problem.

Pseudocode:

AI Approach Pseudo-Code:

1. Main AI Decision Flow:

- First, check if AI has an immediate winning move (oneWinMove)
- If found, return that move
- Else, check if opponent has an immediate winning move (oneWinMove for opponent)
- If found, return that blocking move
- Else, use minimax algorithm with alpha-beta pruning to find best strategic move

2. oneWinMove(board, player):

- For each empty cell adjacent to existing stones:
  - Simulate placing player's stone in that cell

- Check if this move creates 5-in-a-row

- If yes, return this move

- Return None if no winning move found

3. evaluate_board(board, player, easy_mode):

If easy_mode:

- Simply count difference between player's stones and opponent's stones

Else:

- For each stone on board:

  - Check all 8 directions for consecutive stones

  - Count open ends (empty spaces adjacent to sequences)

  - Score based on:

    - 5-in-a-row: +100000 (win) or -100000 (loss)

    - 4 with open ends: high score

    - 3 with open ends: medium score

    - 2 with open ends: low score

    - Weight scores by number of open ends

4. nextmoves(board):

- Find all empty cells adjacent to existing stones

- Return these as possible moves (reduces search space)

5. FUNCTION minimax(board, isMaximizingPlayer, depth, alpha, beta, easy_mode):

// Base Cases:

1. Check if the game has a winner:

   IF checkWinner(board) == AI_PLAYER:

```
    RETURN (None, +∞)  // AI wins (best possible score)
  ELSE IF checkWinner(board) == HUMAN_PLAYER:
    RETURN (None, -∞)  // Human wins (worst possible score)
  ELSE IF depth == 0 OR no valid moves left:
    // Evaluate the current board state
    RETURN (None, evaluate_board(board, AI_PLAYER, easy_mode))

// Generate all possible moves (only near existing stones for efficiency)
possible_moves = nextmoves(board)

// If no moves left (draw)
IF possible_moves is EMPTY:
  RETURN (None, 0)

// Initialize best move and score
best_move = possible_moves[0]  // Default to first move
IF isMaximizingPlayer:  // AI's turn (maximize score)
  best_score = -∞
  FOR each move in possible_moves:
    // Simulate placing AI's stone
    new_board = copy(board)
    new_board[move.row][move.col] = AI_PLAYER

    // Recursively evaluate this move
    _, current_score = minimax(new_board, False, depth - 1, alpha, beta, easy_mode)
```

```
        // Update best move if better score found
        IF current_score > best_score:
            best_score = current_score
            best_move = move

        // Alpha-beta pruning (skip unnecessary branches)
        alpha = MAX(alpha, best_score)
        IF NOT easy_mode AND beta <= alpha:
            BREAK  // Prune remaining moves

    RETURN (best_move, best_score)

ELSE:  // Human's turn (minimize score)
    best_score = +∞
    FOR each move in possible_moves:
        // Simulate placing human's stone
        new_board = copy(board)
        new_board[move.row][move.col] = HUMAN_PLAYER

        // Recursively evaluate this move
        _, current_score = minimax(new_board, True, depth - 1, alpha, beta, easy_mode)

        // Update best move if better score found
        IF current_score < best_score:
            best_score = current_score
            best_move = move
```

```
        // Alpha-beta pruning (skip unnecessary branches)
        beta = MIN(beta, best_score)
        IF NOT easy_mode AND beta <= alpha:
            BREAK  // Prune remaining moves

    RETURN (best_move, best_score)
```

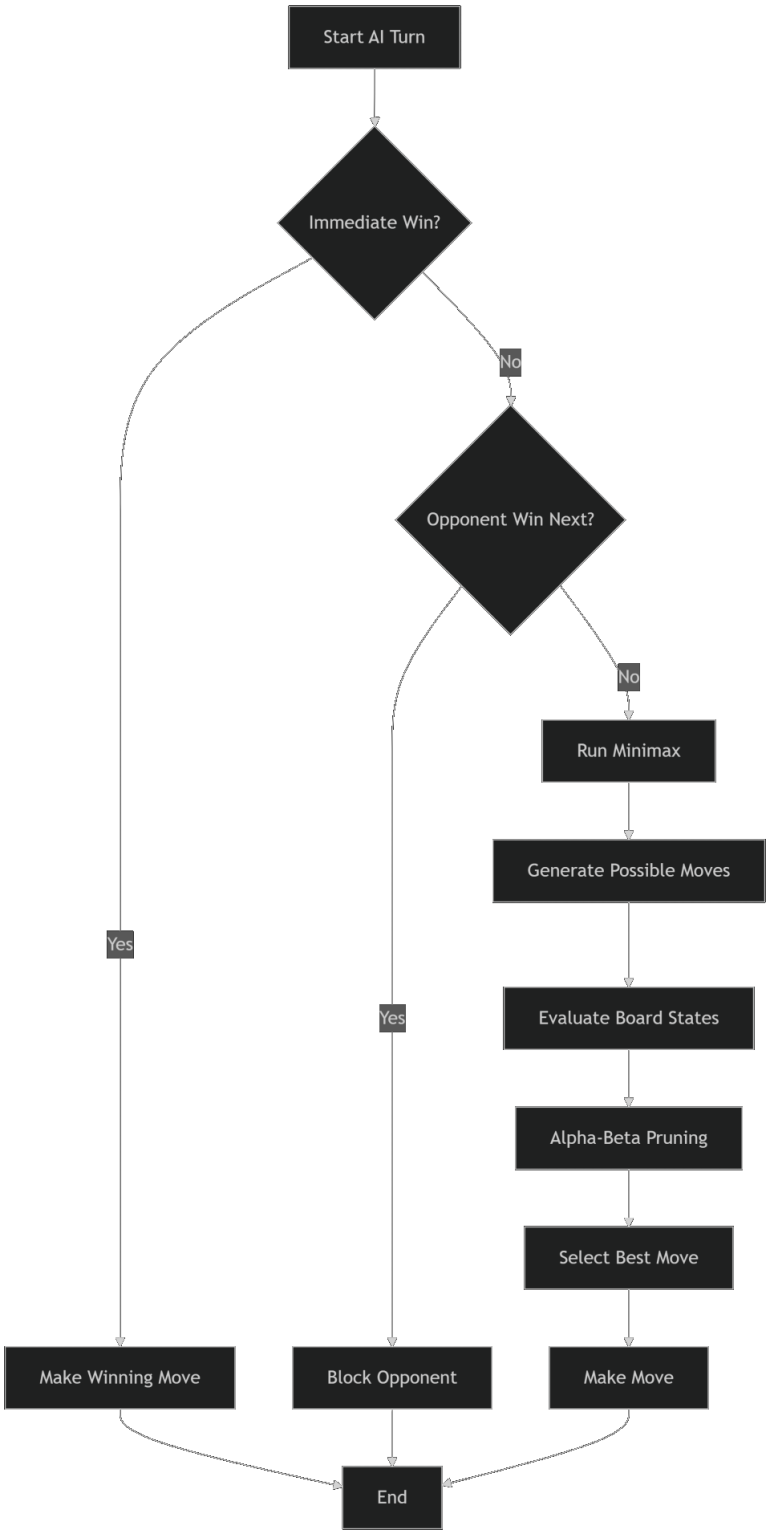
6. AI_Player(board):

- First check for immediate win/block (oneWinMove)
- If none found, use minimax with depth=4 to find best move
- Return the optimal move found

Key Characteristics:

- Uses minimax with alpha-beta pruning for strategic play
- Implements two difficulty levels:
  - Easy: simple material counting
  - Hard: complex pattern recognition
- Focuses search on relevant areas (adjacent to existing stones)
- Prioritizes immediate wins/blocks over strategic play
- Evaluation function heavily weights consecutive stones and open ends

Diagram:



Justifications of Techniques:

1. Why Minimax?

Minimax is a traditional strategy algorithm ideal for two-player, turn-based games such as Gomoku. It ensures the best outcome given that the opponent also plays the best. This makes the AI play as defensively and offensively as possible.

2. Why Alpha-Beta Pruning?

Minus Alpha-Beta pruning, Minimax would be too slow, especially on a 15x15 board with hundreds of possible moves. Alpha-Beta greatly reduces the number of nodes explored to allow deeper lookahead with no loss in performance.

3. Two Heuristics: Why?

We used two heuristics with flexibility as a factor:

- Simple heuristic for faster play and lower difficulty levels.

- Enhanced heuristic to find more problem board opportunities and threats, making a harder 'personality' for the AI with higher difficulty.

4. Why Adjacent Move Optimization?

INSTEAD OF scanning every empty cell (more than 200 at start), we limit our scan to empty cells near placed stones. This cuts down redundant calculations by a lot and guides the AI to reasonable places.

Implementation Details

Programming Language and Libraries:

• Python 3 has been chosen because of simplicity of prototyping and richness in its ecosystem.

• Pygame library has been used to implement an interactive and pleasing graphical user interface (GUI).

• Python's standard libraries including sys and copy have been used for system operation and managing board states.

Code Structure and Explanation:

The code has been divided into several critical components:

1. Game Initialization and Board Creation

At first, the board size (default 15x15), cell size (45 pixels), and window size are established.

A function generateBoard(board) initializes the board as a 2D list of zeros where:

•0 is a free cell,

•1 is the human player stone,

•2 is the AI player stone.

The board can be dynamically resized within the range 5x5 to 19x19 by the user via GUI buttons.

---

2. Drawing Functions (Pygame GUI)

These operations are redrawBoard(window), redrawRocks(window), redrawSidePanel(window, winner), and buttons (increaseButtons, decreaseButtons) are responsible for:

• Drawing the board grid.

• Displaying stones (human with black, AI with white).

• Suggesting the status of the game (whose turn is next or who won).

• Allowing players to reboot the game or re-size the board.

It makes the whole game process highly interactive and sensitive to the players.

---

3. Game Flow and User Interaction

In the top-level game loop:

• MOUSE clicks are heard.

•The human player's move is captured when the human clicks an empty cell.

•The AI makes its move programmatically after the human's move.

•The board is refreshed graphically after every move.

The getClickPosition(pos) function translates mouse positions to board cell positions.

The game continues to alternate turns until a winning player or a board filled with cells.

---

4. Win Checking Logic

To check whether anyone has won:

•The board is searched over and over again after every move.

• checkWinner(board) is the central function that checks for a win horizontally, vertically, diagonally, and anti-diagonally.

• Helper functions checkHorizontal, checkVertical, checkDiagonal, and checkAntiDiagonal search for 5 or more consecutive stones in a sequence.

If a win is detected, the game announces the winner and prevents further moves.

---

5. AI Decision-Making

The AI's moves are determined in multiple stages:

a) Immediate Win or Block

First, before anything else:

• The AI looks for an immediate winning move by applying oneWinMove(board, turn).

• The AI instantly blocks the human opponent if they are going to win.

This keeps the AI highly responsive to threats without excessive searching.

b) Minimax Algorithm with Alpha-Beta Pruning

In the event of no immediate move, the AI uses a more strategic approach:

• The minimax(board, max_turn, depth, alpha, beta, easy_mode) function searches possible future moves.

• It tries to maximize its chances and minimize the opponent's.

• Alpha-beta pruning increases efficiency by removing unpromising branches.

---

6. Heuristic Evaluation (Scoring Board Positions)

The function evaluate_board(board, player, easy_mode) assigns a score to each possible board state.

Depending upon the difficulty mode:

• Easy Mode (Simple Heuristic):

Only monitors the number of stones on each player's board.

• Hard Mode (Advanced Heuristic):

Checks consecutive sequences of stones, open lines, and key formations:

No Longer sequences and open lines have high scores.

Threats are identified and eliminated.

Winning positions are given an extremely high positive value (+100,000) while losing positions are given large negative values (-100,000).

---

7. Optimized Move Generation

Rather than working with each empty cell (a time-consuming prospect), the AI makes use of nextmoves(board) to limit its consideration to empty cells that are adjacent to filled cells.

This greatly reduces computation and allocates attention to areas of the board that count.

---

8. Adaptive Board Resizing

resize functions like resize(window, change) allow dynamic adjustment of board size in-game.

Adjusting board size will reset the game while making sure window changes respond.

Special Features

•Real-Time Gameplay: The AI responds nearly in real-time.

•Board Resizing: Board may stretch or shrink from 5x5 to 19x19.

•Restart Button: Easily restores the game state without having to restart the application.

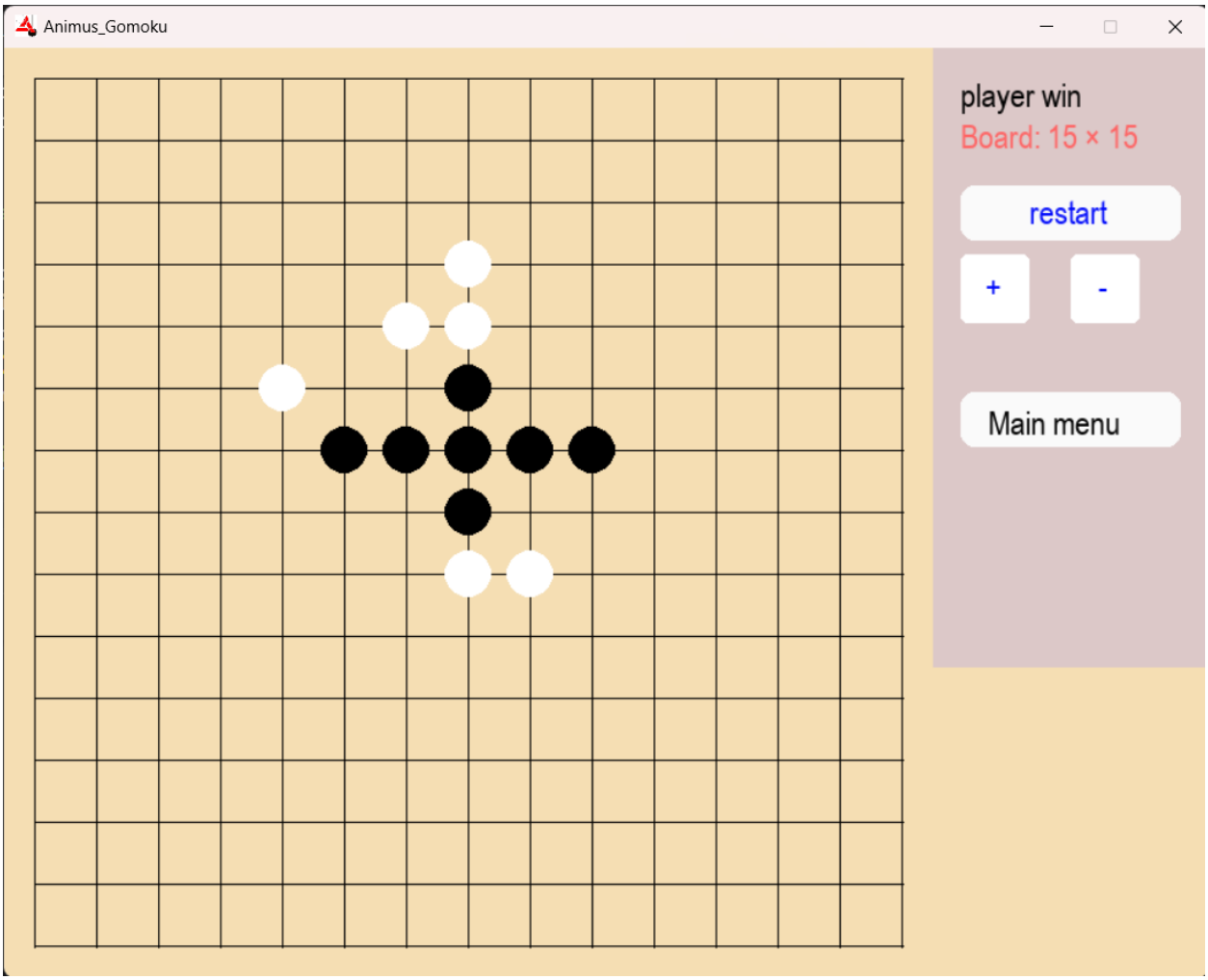
•Clean and User-Friendly UI: Intuitive turn markings and winner alert.

•Performance Optimization: Lessened search trees, shallow depth AI for fast decision-making.

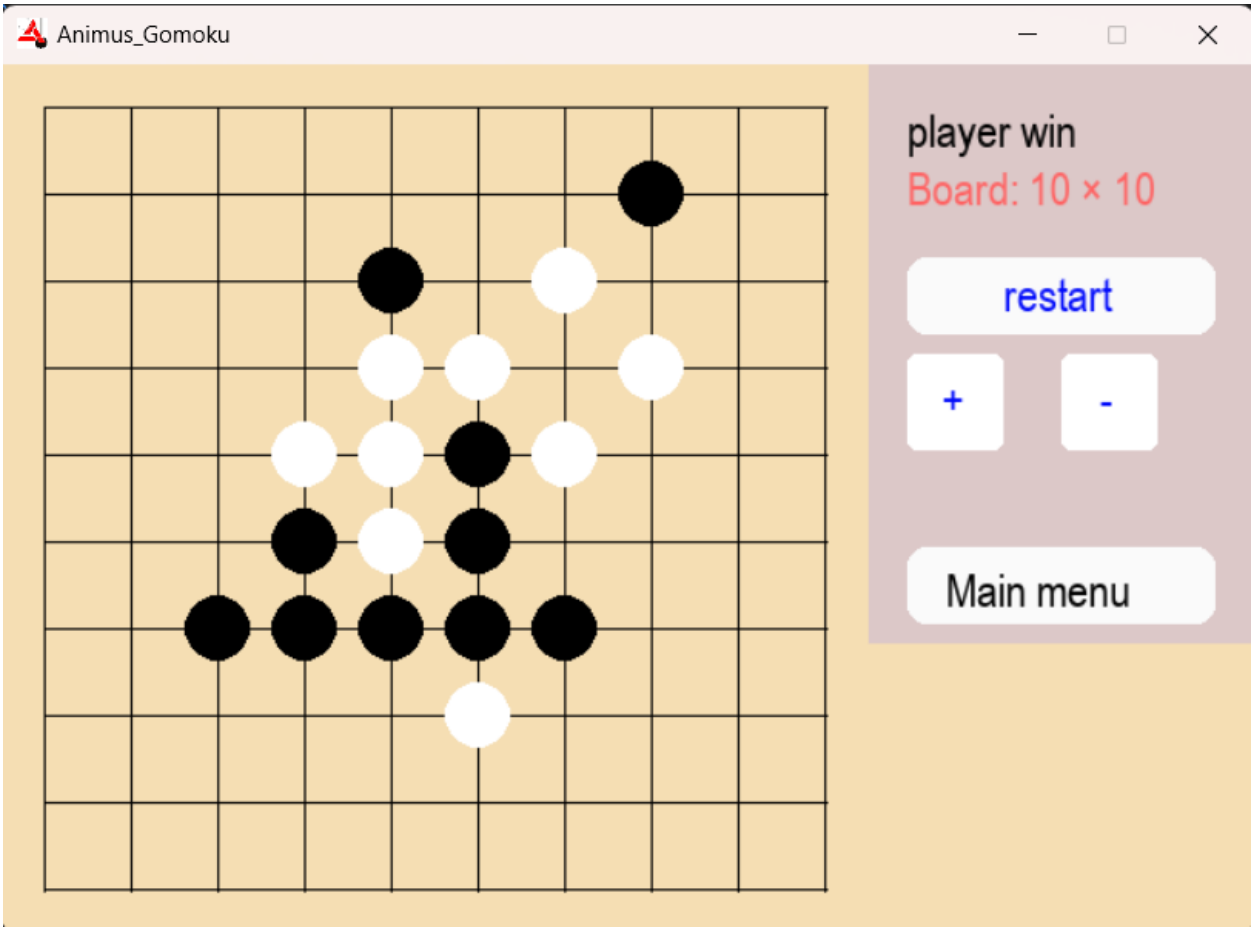
•Two Difficulty Levels: Easy mode for light games, hard mode for serious challenge.

Testing and Evaluation

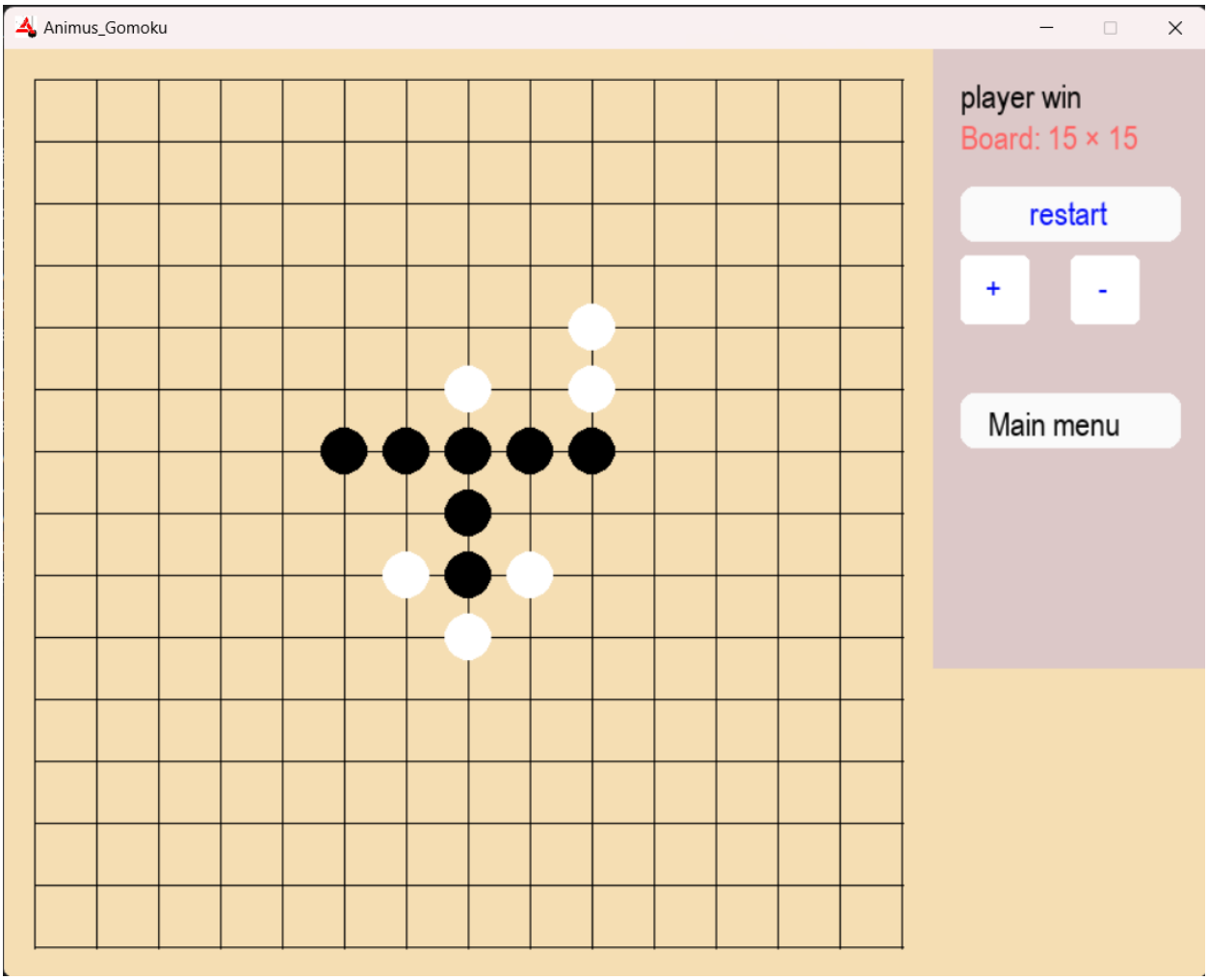
When testing the AI there are 2 things to consider the board size and the game difficulty, so for the first test we ran the game board was set to 15 and the difficulty was set to easy. In the easy difficulty the depth of the search in the minimax function is set to 2 and the evaluation function is very simple letting the AI miss a lot of things which led to the figure below



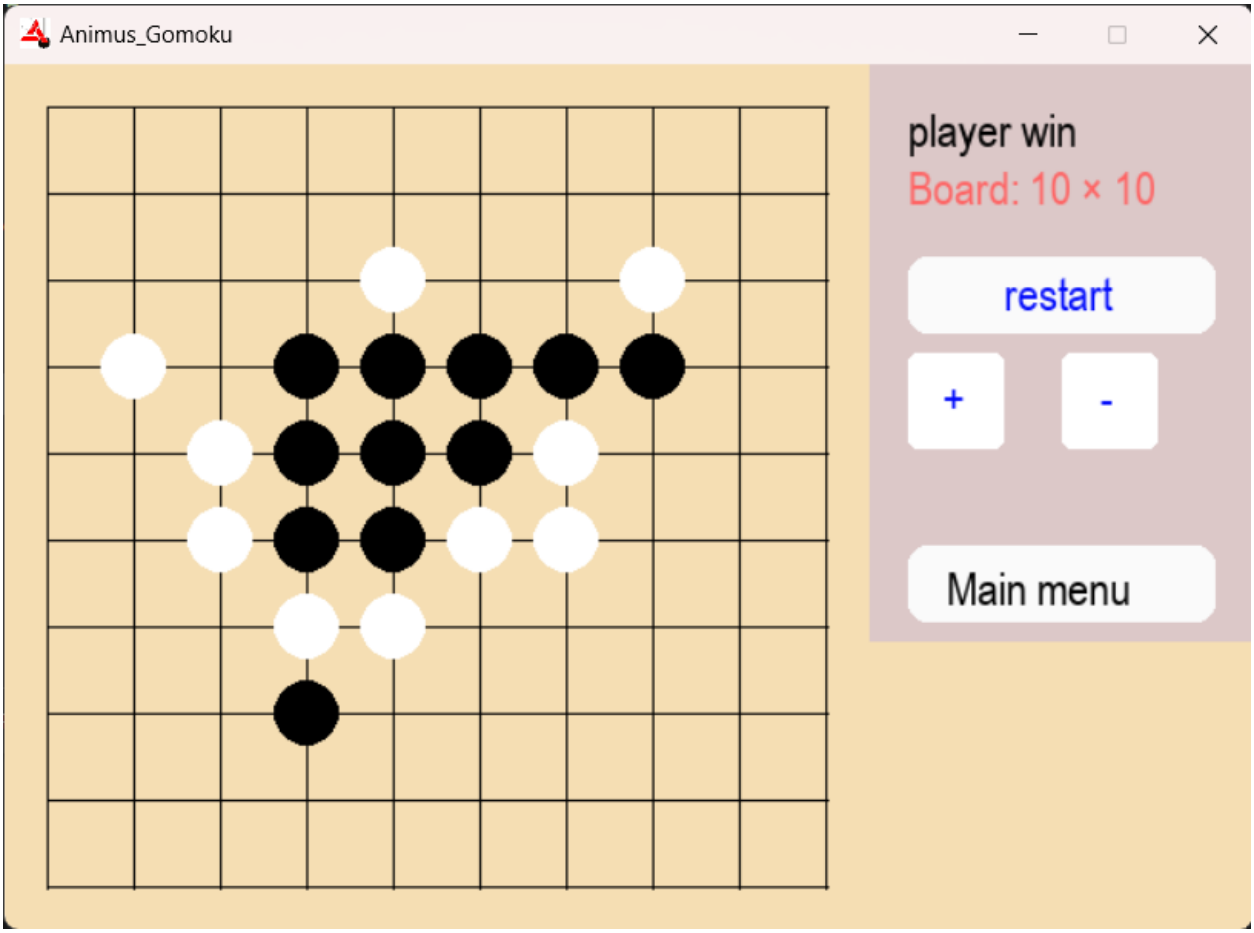
The Second test changed only one thing from the first test the board size was changed from 15*15 to 10*10. After the change happened a lot of things changed aside from the obvious decrease in the number of ways to win the number of cells to check decreases therefore severly decreasing the time taken by the AI to preform the next move but since it was still on easy mode and the depth was the same the result didn't change since it was a very easy win as shown below



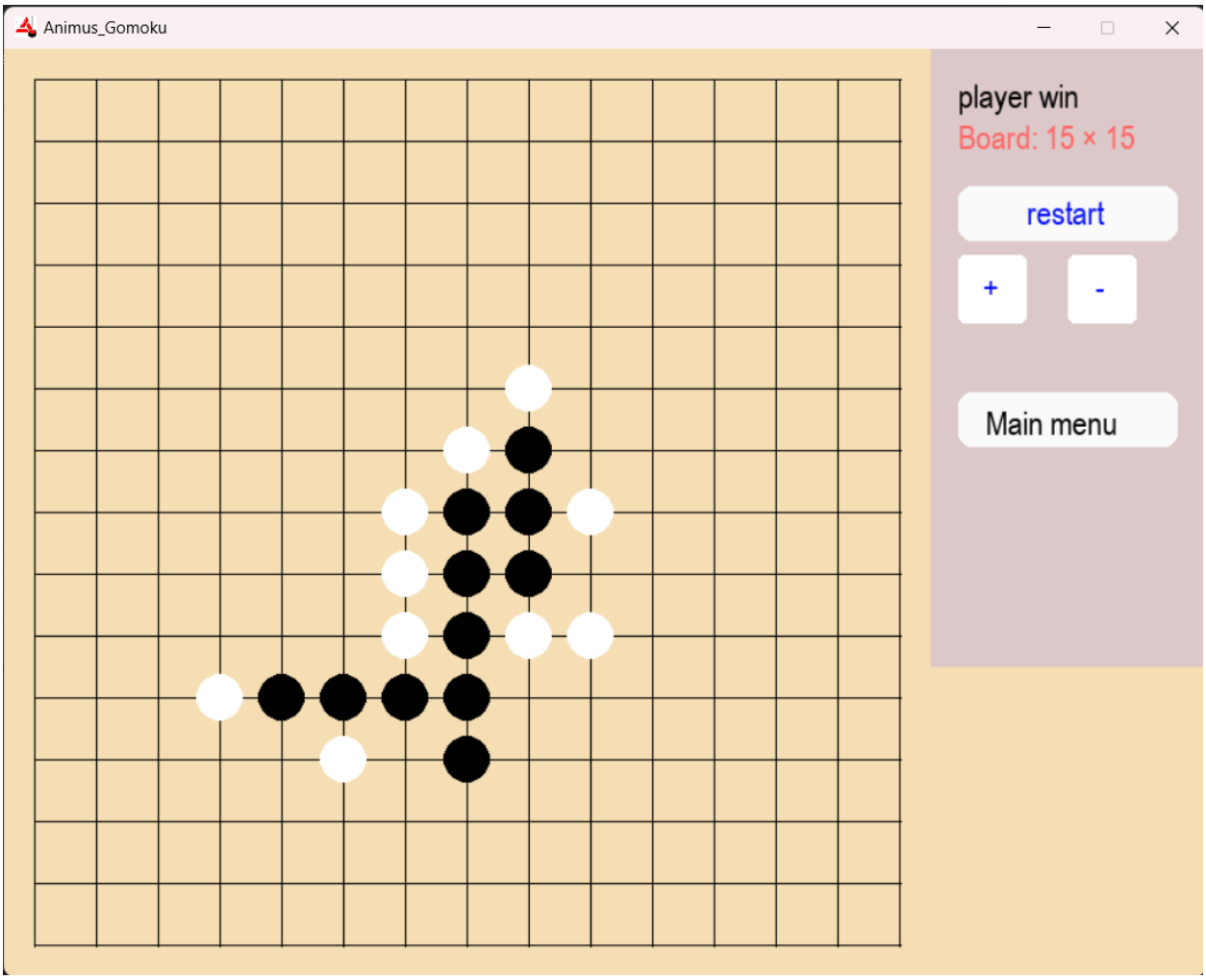
The third test changed game modes from easy to medium which in turn changed the depth from 2 to 3 making the win a bit more difficult. However, the difficulty was still not that hard considering that the only thing that changed aside from the depth is how the evaluation function went from simple counting to complex evaluation considering consecutive stones and open ends. As a result, the AI took a bit more time to evaluate and make the next move. In the end the win was guaranteed as shown below.



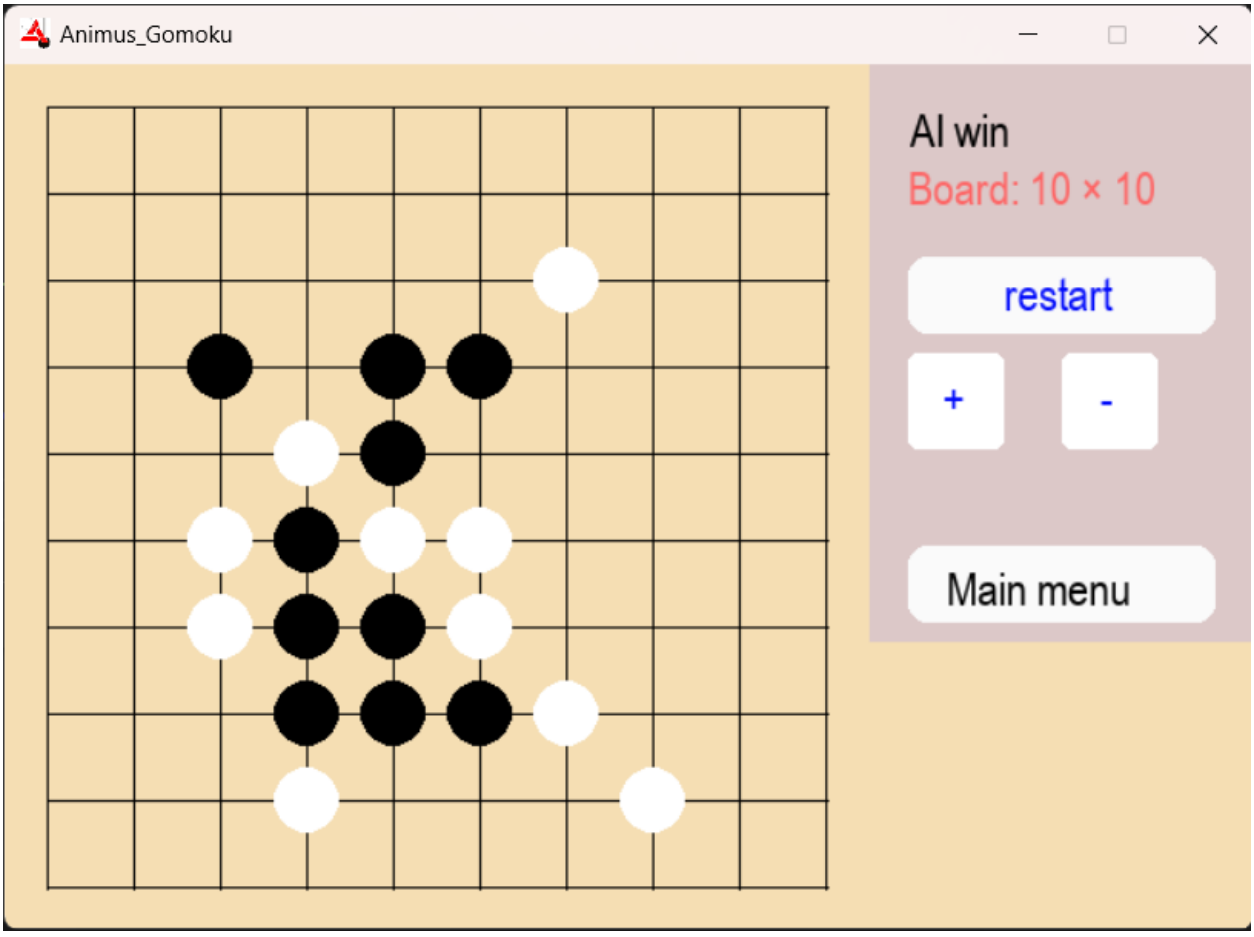
The Fourth test has the same change the second test had from the first test changing only the board size from 15*15 to 10*10 while retaining the depth and difficulty level from the previous test (test 3). After the change the number of permutations greatly decreased allowing the AI to easily evaluate and choose its next move making the win a bit more challenging while taking less time. This, however, didn't change the result of the human victory over AI.



The fifth test will 1 of the most challenging tests since the game mode will be changed to hard mode in this mode the AI will utilize the full capabilities of the minimax function and the alpha beta pruning while also changing the depth of the search from 3 to 4 making the search even more detailed. The hard mode caused a challenge though the win didn't take long the response of the AI was more precise predicting what might happen at every turn and playing accordingly. The AI will start to notice the number of stones standing in a line more efficiently and more accurately. As shown below the AI made the victory path a lot more difficult focusing on when the player can get a win.



The final test involves testing the game at hard mode with a smaller board leaving the AI with less available methods to victory and a higher chance of win. In this test not only is the AI using hard-exclusive functionality like the full capacity of minimax and the function [oneWinMove] which checks everything before every turn to guarantee a hasty victory or a hasty save, it also now has a smaller board meaning the AI will have significantly less combinations of moves to win, which led inevitably to the AI's victory



Challenges and Future Work

When writing the code of the AI to play the game we made sure to add something called critical points these are positions that if played would mean a loss (to the AI), but when implementing the code the AI would not read the critical points of the diagonal win which means when the human player has 4 stones in a line diagonally the AI would not read the critical point where the player can and ignore them. To solve this problem we discovered that in the function Nextmoves that we add 4 more if conditions where he can put a stone at (row+1,col+1), (row-1,col-1), (row-1,col+1) or (row+1,col-1).

Future work would be to add more difficulty levels like how Chess.com has a score to every AI and as the score goes higher the difficulty is higher. Another thing we can add in this project is better testing partners as good as we are at this game we are not beating professional players at this game and if this idea is added we can add more intricate algorithms to further increase the difficulty and professionalism of the AI. The Final think that should be considered is

Reinforced Learning, Go-Moku is a strategy game meaning there can be millions of strategies where a player can end up winning or at the very least strategies that identify not just the counter of the players moves but the best counter, which is a counter that can increase the odds of success for the AI. By using Reinforced Learning the AI can learn from players all over the world gaining new strategies or even creating new ones all together.