Gomuku challenge approach

As the Gomoku game is a solved problem, it's known we need to implement a Minimax search. Further, in order to minimise the search space, I was working on implementation of alpha-beta pruning, and I was planning to implement further iterative deepening in order to limit the depth, with an idea of setting the depth ideally at least for -2 as depth. Based on game flow logic, I believe the depth of 2 levels would be enough to avoid losing, however, ideally, this would be proved by testing.

Further, I was building a simple heuristic function to help the agent to decide the best move.

The main strategy is described above, the details can be read in the pseudocode below. In my package, I had a class minimax with the alpha beta built in into my functions, class board for reading the board with the look around methods and an agent class which would bring the methods together and passing the Move(,).

In order to get the code running further work is needed. My main methods seem to work as intended, I have built a main method and used println to follow how they work. However, when I came to implementing my agent with the given GomokuReferee my main issue seem to be that my agent doesn't know how to read to board. I have tried to fix this issue with different approaches but my agent always ended up seeing an empty board, and passing the same value as the first move for second move as well.

Appendix: Pseudocode

Algorithm 1 Minimax with Alpha- Beta Pruning

```
function initialCall
    getMinValue( board, −∞, ∞)

function getMinValue( board, alpha, beta )
    if board.isTerminal()
        then return evaluateState( board )
    else score = ∞
    for move in board.getLegalMoves() do
        score = getMaxValue( board.createMove(move), alpha, beta )
        beta = Min(beta, score);
        result = Min(score)
        if beta <= alpha then
            break
        return result
function getMaxValue( board, alpha, beta)
    if board.isTerminal() then return evaluateState( board )
    else
    result = −∞
    for move in board.getLegalMoves() do
        score = getMinValue( board.createMove(move), alpha, beta)
        alpha = Max(alpha, score);
```

```
            result = Max(score)
            if beta <= alpha then
                break
        return result
```

Algorithm 2 Simple Heuristic Function

```
function getScoredMoves
    for move in possibleMoves do
        if move does not contain Player then
            //These method calls will add scored moves to a list
            findRow(position);
            findColumn(position);
            findLeftDiagonal(position);
            findRightDiagonal(position);

//Where checkDiagonal changes the diagonal direction
function findMoves(checkColumns, checkRows, checkDiagonal)
    winningScore = 5
    while currentScore != winningScore do
        switch the current column and row if necessary
        if      currentPosition == currentPlayer then
                currentScore = currentScore + 1
        else
                if Not yet complete then
                   switch coordinate traversal direction
                else
                        add to Coordinate possibleMoves
    return score( possibleMoves, currentScore )
function checkLeftDiagonal(move) findMoves(true, true, false)
function checkRightDiagonal(move) findMoves(true, true, true)
function checkRows(move) findMoves(true, false, false)
function checkColumns(move) findMoves(false, true, false)
```

Algorithm 3 Depth limiting in the MiniMax algorithm

```
function initialCall
    getMinValue( board, $-\infty$, $\infty$, 4)

function getMinValue( board, alpha, beta, depth )
    if      board.isTerminal() || depth == 0 then
            return evaluateState( board )
    else
            for move in board.getLegalMoves() do
```

```
                score = getMaxValue(
                board.createMove(move),
                alpha,
                beta,
                depth - 1
                )


function getMaxValue( board, alpha, beta, depth )
    if      board.isTerminal() || depth == 0 then
            return evaluateState( board )
    else
            for move in board.getLegalMoves() do
                score = getMinValue(
                board.createMove(move),
                alpha,
                beta,
                depth - 1
                )
```

Algorithm 4 Scoring MiniMax moves

```
function scoreState
    Heuristically evaluates state
    result = gameOver && winner == "Draw" ?
                0 : evaluateMove()
    result = winner.isOpponent() ?
                ( result + depth ) - 100 :
                ( result + 100 ) - depth
```