AI PRINCIPLES & TECHNIQUES

Four In A Row Assignment Report

Ege SARI *s1034535*
Radboud University *13 October 2021*

Contents

1 Introduction

Introduction

1.1 Goals of the Project

Some search problems can be represented in a tree form, especially the problems for game search. The MinMax algorithm is often used for this search goal. It is a decision algorithm formulated for n-player zero-sum game theory To optimize it, most of the time it is used with alpha-beta pruning which is a technique used to prune illogical movements from tree.
In this assignment, we are asked to implement a tree structure to store game states, and then implement the MinMax algorithm with and without alpha-beta pruning. Then we are asked to discuss the time complexities of both versions of the MinMax algorithm, with and without pruning.

1.2 About the Game

The game we are playing is known as N-in-a-row. It is a game where each player wants to make N tokens in a row, vertically, horizontally or diagonally. It is a zero-sum game for 2 players. The first player use the token X and the second player use the token O.

2 Design

In this section we will explain the design of the structures and algorithms we used for this project. Also we will present their classical way as an alternative. We will give the pseudo-code form of the algorithms and structures.

2.1 Design of Tree Structure

A tree consists of nodes and leaves where a node has child(ren) nodes or leaves, and a leaf is a bottom part of the tree with no child. In a game tree, if the wanted depth is reached or if the game is over with a final state (player 1 or player 2 wins, or it is a draw), the node is a leaf and it has no child further. In other cases, if the game is unfinished and the maximum depth is not reached, then it is a node with child(ren). The node contains the information about the game : the board state, the player, and its child(ren) if possible. The example of a game tree can be seen in Figure 1 where the game has a board of size 4x1.

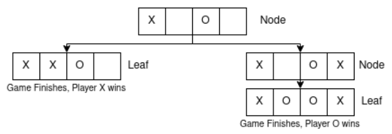

Figure 1: An example of a game tree.

To build a tree, a recursive structure is used often. This recursion returns a node with either child(ren) or no child, which is a leaf. If the maximum depth is reached or the game is over, then it is a base case and returns the node and the algorithm stops. If the depth is not reached and the game is not over, then for every possible move that player can do, tree calls itself recursively. For this time the algorithm gets new arguments such as a new board with that move made, the next player and the depth decremented by one. And this tree again is added to the children list of the node and the node is returned. The pseudo-code of the algorithm is illustrated in Algorithm 1. It is useful to understand this structure.

Algorithm 1 Building Tree

```
procedure BUILDTREE($board, player, depth$)    ▷ The game tree starts with the board and player
    node← $Node(board, player)$                ▷ Initialize a new node
    if board.gameIsNotOver and depth !=0 then  ▷ It is a Node
        for each possible move in column i do
            newBoard← $Board(i, player)$       ▷ clone a new board with move
            nextPlayer← if player =1 then 2 else 1    ▷ create nextPlayer
            child← $buildTree(newBoard, nextPlayer, depth - 1)$    ▷ Recursive case
            node.children.add(child)           ▷ add this child to the children of node
        end for
    end if
    return node    ▷ if the created node is a leaf then, it will have no child added
end procedure
```

Alternatively there can be an attribute of Node named depth, so that each we can determine the depth of every node. But I didn't and I explained the reason of my choice in the next section, 2.2. But the classical way, Node structure with depth attribute can be found in Appendix section with its pseudo-code and possible implementation.

2.2 Design of MinMax Algorithm

The principle of MinMax algorithm is simple : The Maximizing Player (in our case it is player 1, X) wants to maximize its gain and the Minimizing Player (in our case it is player 2, O) wants to minimize the maximizing player's gain. In MinMax algorithm, the Maximizing Player looks for its children with again applying MinMax recursively and returns the maximum value among them. The Minimizing Player looks for its children as same as the Maximizing Player does, but it returns the minimum value among them. If the node has no child which means the game is over or the maximum depth is reached, it returns its heuristic value. This is the basic explanation of MinMax algorithm.
In details, we can explain the algorithm in three cases. The first one is the base case, if the player node is only leaf, then it returns the heuristic value of the board it has. If the player is not a leaf and if it is the Maximizing Player, a value called maximum value is initialized with its initial value negative infinity. It is initialized with negative infinity because we want to return the maximum value that it can have. Then for each child of that node, MinMax algorithm is applied on them recursively, by returning their MinMax value. For each MinMax value from child a comparison is made between that value and maximum value initialized at the beginning. If the MinMax value is higher the maximum value is updated and the algorithm continues with the next child (if possible). If it is not higher, then maximum value is not updated and the algorithm continues with the next child. After all children are checked, the maximum value is returned. If the player is Minimizing Player, we do the same thing as we did with Maximizing Player, with some differences. A minimum value is initialized with positive infinity since we want to return the minimum value. Again the comparison is made between the MinMax value of the children and the minimum value but now, if the MinMax value is less then minimum value, then the minimum value is updated. At the end the minimum value is returned. The pseudo-code is illustrated in Algorithm 2.

Algorithm 2 MinMax without pruning

```
procedure minMax(node)
    if node.children.isEmpty() then                ▷ If node has no child then it is a leaf
        return heuristic(node.board, node.player)    ▷ return heuristic evaluation
    else
        if node.children == MaximizingPlayer then
            maximumValue ← $-\infty$
            for every child c of node do
                value← minMax(c)
                maximumValue ← maxOf(value, maximumValue)
            end for
            return $maximumValue$
        else                                         ▷ It is minimizing player
            minimumValue ← $+\infty$
            for every child c of node do
                value← minMax(c)
                minimumValue ← minOf(value, minimumValue)
            end for
            return $minimumValue$
        end if
    end if
end procedure
```

The classical way of the algorithm uses depth as a parameter. I didn't used it because I didn't need it. I should have changed the structure and add a new attribute called depth and every node must save its own depth. But I can understand that if the maximum depth is reached, the node will have no child. So I use the number of children of a node in order to check if the depth is reached or the game is over. This is the reason why I have chosen this design. The pseudo-code and possible implementation of classical design of MinMax algorithm can be found in the Appendix section.

2.3 Design of Alpha-Beta Pruning

The alpha-beta pruning is a simple pruning technique to improve the MinMax algorithm. We use two additional values namely "alpha" for maximizing player and "beta" for minimizing player.In the maximizing player, if the MinMax value of the child is bigger than the maximum value, the maximum value is updated. Then we check if the updated maximum value is bigger than beta value. If it is then we break the loop and stop looking for other child nodes, we prune the next branches. If it is not, then we update the alpha value if it is less than the maximum value. The same applies for minimizing player. If the MinMax value of the child is smaller than the minimum value we update the minimum value. Then we check if the updated minimum value is less than alpha value. If it is then we break the loop and stop looking for other child nodes, we prune the next branches. If it is not, then we update the betaalpha value if it is less than the minimum value. It will be better understood in pseudo-code in Algorithm 3.

3 Complexity

In this section we are going to show, prove and discuss the complexity of MinMax algorithm with and without Alpha-Beta pruning.

Algorithm 3 MinMax with Alpha-Beta Pruning

```
 1: procedure ALPHA-BETA($node, alpha, beta$)
 2:     if node.children.isEmpty() then            ▷ If node has no child then it is a leaf
 3:         return heuristic(node.board, node.player)      ▷ return heuristic evaluation
 4:     else
 5:         if node.children == MaximizingPlayer then
 6:             maximumValue $\leftarrow -\infty$
 7:             for every child c of node  do
 8:                 value$\leftarrow$ minMax(c, alpha, beta)
 9:                 maximumValue $\leftarrow$ maxOf(value, maximumValue)
10:                 if beta $\leq$ maximumValue then break
11:                 end if
12:                 alpha $\leftarrow$ maxOf(maximumValue, alpha)
13:             end for
14:             return $maximumValue$
15:         else                                          ▷ It is minimizing player
16:             minimumValue $\leftarrow +\infty$
17:             for every child c of node  do
18:                 value$\leftarrow$ minMax(c,alpha, beta)
19:                 minimumValue $\leftarrow$ minOf(value, minimumValue)
20:                 beta$\leftarrow$ minOf(value, beta)
21:                 if minimumValue $\leq$ $alpha$ then break
22:                 end if
23:                 beta$\leftarrow$ minOf(minimumValue, beta)
24:             end for
25:             return $minimumValue$
26:         end if
27:     end if
28: end procedure
```

3.1 Complexity of MinMax Algorithm Without Pruning

We can consider MinMax algorithm is a recursion which returns the maximum utility value of a node from MinMax value of its children nodes based on their utility. So, MinMax algorithm works with the same principle as Depth First Search (DFS) does. It looks for the first child of every node every time and it continues to apply MinMax recursively until it reaches the bottom of the tree. So we can say that MinMax algorithm has the same complexity as DFS has for implicit graphs : $\mathcal{O}(c^d)$ where c is the branching factor i.e. the number of children a node has and d is the depth of the tree.
But we should also show the complexity by proving it is indeed has the complexity of $O(c^d)$. Based on the pseudo-code in Algorithm 2, for base case, $T(1)$ we have :

- heuristic(node.board, node.player) = $\mathcal{O}(heuristic)$

where $\mathcal{O}heuristic)$ represents the complexity of the heuristic function. The calculation of the complexity of the heuristic is out of scope for this project. But we can assume this as a constant. And for recursive case of maximizing player,$T(d)$ we have :

- maximumValue $\leftarrow -\infty = \mathcal{O}(1)$
- value$\leftarrow$ minMax(c) $= c \cdot T(d-1)$
- maximumValue $\leftarrow$ maxOf(value, maximumValue) $= \mathcal{O}(c)$

where d is the depth of the tree and c is the number of children the node has. The total complexity for maximizing player, we have $T(d) = \mathcal{O}(1) + c \cdot T(d-1) + \mathcal{O}(c) = c \cdot T(d-1) + \mathcal{O}(c+1)$. The same applies for the minimizing player but only one of them can be chosen at one time so we can write our recursion complexity as :

$$\begin{aligned}
&T(d) = c \cdot T(d-1) + \mathcal{O}(c+1) \\
&= c^2 \cdot T(d-2) + \mathcal{O}(c^2 + 2c + 1) \\
&= c^3 \cdot T(d-3) + \mathcal{O}(c^3 + 3c^2 + 3c + 1) \\
&\vdots \\
&T(d) = c^k \cdot T(d-k) + \mathcal{O}((c+1)^k) \\
&T(d) = c^d \cdot T(0) + \mathcal{O}((c+1)^d) \qquad \text{for } d = k
\end{aligned}$$

We can assume that $T(0) = 1$ since it is constant for returning the value of heuristic function, so we have the complexity of $\mathcal{O}(c^d + (c+1)^d) = \mathcal{O}(c^d)$ where c is the number of the children, the branching factor (in our project it is width of the board) and d is the depth.
By this we have proven that MinMax algorithm has the complexity of $\mathcal{O}(c^d)$.

3.2 Complexity of MinMax Algorithm With Pruning

The worst case complexity of the MinMax algorithm with alpha beta pruning is again the same $\mathcal{O}(c^d)$. Because in the worst case, there will be node branch pruned so again the MinMax algorithm will be applied to every node in the tree. In the best case of the MinMax algorithm with alpha beta pruning, the first player can find the maximum value in the first child, and the second player will look for every child which will take c times. So if it will take $\mathcal{O}(c^{d/2})$ since we will look for every child in every two level.

3.3 Comparison Between the Complexities

It can be seen that alpha-beta pruning can improve the complexity of $\mathcal{O}(c^d)$ to the complexity of $\mathcal{O}(c^{d/2})$ which can be $c^{d/2}$ faster than the algorithm without pruning. We will also discuss this improvement based on the tests in Section 5.

4 Implementation

In this section, the implementation choice is explained and the implementation is illustrated.

4.1 Implementation of Tree Structure

The tree consist of two important structure : The node structure and buildTree function.

4.1.1 Node Structure

For Node, I have chosen not to implement a depth attribute. The main reason for this is the lack of need. The depth attribute is used for MinMax algorthm with depth parameter, where that parameter is used to check if the node is at the bottom with the maximum depth or not. As it is shown in detail in Section 4.2, I do not use a depth parameter for MinMax algorithm. The Node structure has three attributes : board for the board state, player for the player ID and children for its children nodes. The implementation is given in below :

```
public class Node {

        public Board board;
        public List<Node> children;
        public int player;

        Node(Board board, int player) {
            this.board = board;
            this.player = player;
            children = new ArrayList<>();
        }
    }
```

4.1.2 Building Tree

The method buildTree is located in Tree class. The class has an attribute named ”root” which is used in constructor method by calling buildTree() method. It can be seen below :

```
public class Tree {

    public Node root;

    public Tree(Board init_board, int gameN, int playerId, int depth) {
        this.root = buildTree(init_board, playerId, gameN, depth);
    }
}
```

For the method buildTree() I have used a recursive structure such that it will recursively call itself it he depth is not reached and it is not a leaf. There are some remarks on the implementation :

- I initialize a node with the current board and player

- I have defined a boolean named isOver by using the returned value of Game.winning() function. By this I was able to check if the game is over or not
- If the game is over or the maximum depth is reached, I return the initialized node.
- If the game is not over or the maximum depth is not reached, for every possible moves, I create a new board by cloning the current game board. Then I apply play function on it, which is a function that checks if the move is legal and then it plays the move on the board and changes the board state.
- I create an integer value called nextPlayer to use it in the recursive call, to represent the next player.
- I declare the child node with the recursive all and add this object to the children list of the node.
- At the end I return the node.

The implementation can be seen below :

```java
public Node buildTree(Board board, int playerId, int gameN, int depth) {

        Node node= new Node(board, playerId);
        int winner = Game.winning(board.getBoardState(), gameN);
        boolean isOver = winner != 0;
        if (!(isOver || depth==0))
        {
            for (int i = 0; i < board.width; i++) {
                Board cloneBoard = new Board(board);
                if (cloneBoard.play(i, playerId)) {
                    int nextPlayer = (playerId == 1) ? 2 : 1;
                    Node child = buildTree(cloneBoard, nextPlayer, gameN, (depth-1));
                    node.children.add(child);
                }
            }

        }
        return node;
    }
```

4.2 Implementation of MinMaxPlayer

The MinMaxPlayer consists of two important structures. One of them is the method makeMove and the other one is the method minMax.

4.2.1 makeMove Method

The method is used to return the column number with the best utility. There are some remarks about this implementation:

- I initialize a maximum value with the smallest integer value and a column number with zero.
- for each legal move I create a new board and a new tree based on that board.

- I return the MinMax value of that tree. If it is better than the maximum value I update the maximum value and I update the column number
- after all children is looked, I return the column number with the maximum utility.

The implementation can be seen below :

```java
public int makeMove(Board board) {
        System.out.println(board);
        if (heuristic != null)
        {
            int maxValue = Integer.MIN_VALUE;
            int maxMove = 0;
            for(int i = 0; i < board.width; i++)
            {
                if(board.isValid(i)) {
                    Board newBoard = board.getNewBoard(i, playerId);
                    Tree gameTree = new Tree(newBoard, gameN, playerId,depth);
                    int minMaxvalue = minMax(gameTree);
                    if(minMaxvalue > maxValue)
                    {
                        maxValue = minMaxvalue;
                        maxMove = i;
                    }
                }
            }
            System.out.println("Heuristic: " + heuristic + " calculated the best move is: "
            + (maxMove + 1));
        }
        System.out.println("Player " + this + "\nWhich column would you like to play in?");
        int column = scanner.nextInt();
        System.out.println("Selected Column: " + column);
        return column - 1;
    }
```

4.2.2 minMax Method

The method is used to return the MinMax value of a board. It was implemented in two methods, one gets a tree as parameter and used in makeMove() method and the other one gets a node as parameter and used for recursion. There are some remarks about this implementation :

- As it was explained in previous sections, I didn't use a depth parameter for this function. The reason was explained in Section 4.1.1, briefly there was no need. Instead of depth parameter, I check if the bottom is reached by checking the children number of the node. If the children list of the node is empty, then we can be sure that the depth is reached or the game is over. We can prove that this holds because of the algorithm design of buildTree() method.
- The method is indeed structured as the algorithm is designed, can be seen in Algorithm 2.
- If the depth is reached or the game is over, then the method returns the heuristic value of the board. Here I have used evaluateBoard() method from Heuristic class.

- If the game is not over or the depth is not reached, then I check the player by using the player attribute of node object. I use a constant value named "Player_MAX" which is an integer with value 1.
- If the player ID equals to 1, that means it is the first player and maximizing player. Then recursively for each child of the node, again the MinMax is applied and returned the maximum value of them.
- If the player ID is not equal to 1, that means it is the second player and minimizing player. Then recursively for each child of the node, again the MinMax is applied and returned the minimum value of them.

Below the implementation is illustrated :

```
    public int minMax(Tree tree)
    {
        return (minMax(tree.root));
    }

    private int minMax(Node node) {
        if (node.children.isEmpty())
        {
            return heuristic.evaluateBoard(node.player, node.board);
        } else {
            if (node.player == Player_MAX)
            {
                int maximumVal = Integer.MIN_VALUE;
                for (Node child : node.children)
                {
                    int value = minMax(child);
                    maximumVal = Math.max(value, maximumVal);
                }
                return maximumVal;
            }
            else
            {
                int minimumVal = Integer.MAX_VALUE;
                for (Node child : node.children)
                {
                    int value = minMax(child);
                    minimumVal = Math.min(value, minimumVal);
                }
                return minimumVal;
            }
        }

    }
```

4.3 Implementation of Alpha-BetaPlayer

The Alpha-BetaPlayer consists of two important structures like MinMaxPlayer and there are some differences between them. In this section, the differences are explained

4.3.1 makeMove Method

There is only one difference between the methods of MinMaxPlayer and AlphaBetaPlayer. The difference is we assign minMaxvalue to alpha-beta version of MinMax with initial values of alpha equals to the smallest integer value and beta equals to the biggest integer value :

```java
public class AlphaBetaPlayer extends PlayerController {

    final int Alpha_init = Integer.MIN_VALUE;
    final int Beta_init = Integer.MAX_VALUE;
```

```java
    int minMaxvalue = minMax_Alpha_Beta(gameTree,Alpha_init, Beta_init);
```

4.3.2 minMax_Alpha_Beta Method

The difference is also explained in Section 3.2 with Algorithm 3. I have implemented the same algorithm in Java. Needless to explain the algorithm and implementation again, I should focus on the differences with minMax() method.

- The method gets two extra arguments for alpha value and beta value
- If the player is maximizing player, we make a comparison between maximumVal and beta. If beta is bigger we stop the loop for looking for the next children which is the next branches with break command. If not we update the alpha value with maximum value and keep looking for the next child if possible :

```java
                    if(maximumVal <= beta) {
                        break;
                    }
                    alpha = Math.max(maximumVal, alpha);
```

- If the player is minimizing player then we make a comparison between minimumVal and alpha. If alpha is bigger we stop the loop for looking for the next children which is the next branches with break command. If not we update the beta value with minimum value and keep looking for the next child if possible:

```java
                    if(minimumVal <= alpha) {
                        break;
                    }
                    beta = Math.max(minimumVal, beta);
```

5 Testing

In this section, the test results of the implementations are reported. The test are based on four things : The players, the goal number N, the board size and the depth of the tree.

For the players we will use 4 cases : HumanPlayer vs. MinMaxPlayer, MinMaxPlayer vs. MinMaxPlayer, HumanPlayer vs. Alpha-BetaPlayer and Alpha-BetaPlayer vs. Alpha-BetaPlayer. For N we will use 2 cases : N= 2 and N=4. For the board size we will use 2 cases : 7x6 and 4x2. For depth we will use 2 cases : depth =2 and depth =6. There will be 32 tests in total. I will play the move what the algorithms decided and I will measure how many times the board is evaulated. Of course, I expect Alpha-BetaPlayer will evaluate less because of the pruning. In the first and third parts, I will test the algorithms about their strength for preventing my win. The second and fourth part will test their strength for not only preventing the win, but also winning the game. For convenience I will indicate tests and the inputs in format : Test x (s, N , d) where x is the test number, s is the size of the board, N is the goal number and d is the depth. The results will be discussed in Section 6.

5.1 HumanPlayer vs. MinMaxPlayer

This part, will be about the strength of the MinMax algorithm for preventing me, the first player to win the game.

Test 1 (4x2, 2, 2)
The game finishes in 4 moves averagely and MinMax algorithm evaluate the board 55-58 times on average. I, the human player can win most of the games.

Test 2 (4x2, 2, 6)
The game finishes in 4 moves averagely and MinMax algorithm evaluate the board 30-32 times on average. Again the human player, I can win the most if the games, but compared the previous test, it seems the second player, MinMaxPlayer makes move to prevent my victory.

Test 3 (4x2, 4, 2)
The game finishes in 7 moves averagely and MinMax algorithm evaluate the board 65 times on average. Surprisingly it tried to block my way even the depth is too low. And there were some games finishes with draw.

Test 4 (4x2, 4, 6)
The game finishes in 8 moves averagely and MinMax algorithm evaluate the board 660 times on average. Again it was a test if indeed the algorithm can prevent me or not. It was a bit unsuccessful, I assume because of the heuristic evaluation of the board. Even with the given depth, the algorithm didn't work as I expected, it didn't try to prevent me by blocking my way and it was not better than the previous test.

Test 5 (7x6, 2, 2)
The game finishes in 4 moves averagely and MinMax algorithm evaluate the board 310 times on average. It behaves as I expected, does not try to prevent me, the first player much, since the depth is too low. But if I make a mistake i.e. skip the chance to win, it wins.

Test 6 (7x6, 2, 6)
The game finishes in 4 moves averagely and MinMax algorithm evaluate the board 22.000-26.000 times on average. It behaves a bit worser than I expected. Again it is not so capable to block my winning but it seems it knows how to win.

Test 7 (7x6, 4, 2)

The game finishes in 4 moves averagely and MinMax algorithm evaluate the board 1100 times on average. It behaves as I expected, as the other tests with lower depths. Indeed it plays like thinking the optimal in one step ahead. But if I make a mistake i.e. skip the chance to win, it can't win since the goal is higher.

Test 8 (7x6, 4, 6)
The game finishes in 8-9 moves averagely and MinMax algorithm evaluate the board 280.000 times on average. For some moves, it wants to prevent me, it is better than the previous test with lower depth, but it is not so successful as I thought. Sometimes it behaves blindly, it does not prevent me even with my next move I can win. I am not sure about this, maybe again it is because of the heuristics.

5.2 MinMaxPlayer vs. MinMaxPlayer

This part, will be about the strength of the MinMax algorithm for both preventing and winning.

Test 1 (4x2, 2, 2)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 41 (for first player) and 29 (for second player) times. The first player wins. Unfortunately there is nothing much to do for the second player to prevent the first player winning.

Test 2 (4x2, 2, 6)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 41 (for first player) and 29 (for second player) times. At first I didn't expect to get the same evaluation number but since the goal number N is so small even the depth is higher, the game usually ends at depth 2.

Test 3 (4x2, 4, 2)
The game finishes in 7-8 moves averagely and MinMax algorithm evaluate the board 89 (for first player) and 72 (for second player) times. Unfortunately the second player does not make much to prevent the first player.

Test 4 (4x2, 4, 6)
The game finishes in 8 moves averagely and MinMax algorithm evaluate the board 2600 (for first player) and 700 (for second player) times. The program has a fault on returning the column. It returned some invalid column numbers. I am not sure about the source of this fault. The significant difference is X wants to start with the second column where it has higher chance to win rather than starting with the first column

Test 5 (7x6, 2, 2)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 404 (for first player) and 229 (for second player) times. It behaves as I expected, since both the depth and the goal number is so low, the second player can't do anything for.

Test 6 (7x6, 2, 6)
The game finishes in 3-4 moves averagely and MinMax algorithm evaluate the board 36.000(for first player) and 13.000 (for second player) times. It behaves cleverly for both sides. So yes, unfortunately the second player can not do much since the goal number is low, but if the first player skip to win, then second player wins the game.

Test 7 (7x6, 4, 2)
The game finishes in 18-20 moves averagely and MinMax algorithm evaluate the board 2250(for first player) and 1600 (for second player) times. The players want to fill the column up, because of the heuristics, I suppose. Even the depth is low, the second player can make some prevention moves sometimes. But it is not optimal.

Test 8 (7x6, 4, 6)
The game finishes in 18-20 moves averagely and MinMax algorithm evaluate the board 350.000 (for both sides) times. It behaves as I expected, the players want to fill the column up, again because of the heuristics, I suppose. But it can be seen that if there are random moves made (i.e. I don't play as the algorithm wants) then the second player wants to prevent the first player and first player finds the optimal solution indeed. But as it can be seen, the evaluation is high. Another remark, again the first player starts with the second column to get higher chance.

5.3 HumanPlayer vs. Alpha-BetaPlayer

This part, will be about the strength of the Alpha-Beta algorithm for preventing me, the first player to win the game.

Test 1 (4x2, 2, 2)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 13-17 times on average. I, the human player can win most of the games.

Test 2 (4x2, 2, 6)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 8-10 times on average. Again the human player, I can win the most if the games, but it makes less evaluation compared to the previous test. Actually I was not expecting this, since the depth is too low, I thought there will be no difference at all. But in fact, it nearly halved the evaluation time. We can understand that our implementation works, but it is best to continue with test cases

Test 3 (4x2, 4, 2)
The game finishes in 7 moves averagely and MinMax algorithm evaluate the board 41 times on average. It does not try to prevent my move but since the depth is too low it can be understandable.

Test 4 (4x2, 4, 2)
The game finishes in 7-8 moves averagely and MinMax algorithm evaluate the board 28-35 times on average. But there is a problem, I assume about the program itself, even the column is not empty, it recommended me to play on that column.

Test 5 (7x6, 2, 2)
The game finishes in 3-4 moves averagely and MinMax algorithm evaluate the board 76-80 times on average. It behaves as I expected, does not try to prevent me, the first player much, since the depth is too low. But if I make a mistake i.e. skip the chance to win, it wins.

Test 6 (7x6, 2, 6)
The game finishes in 3-4 moves averagely and MinMax algorithm evaluate the board 60-70 times on average. The interesting part is, If I let the game play be longer, the

evaluation becomes a bit less.

Test 7 (7x6, 4, 2)
The game finishes in 13-14 moves averagely and MinMax algorithm evaluate the board 480-520 times on average. It is not good, I assume because of the depth. It tries some prevention moves also some winning moves but it is same as the MinMax without pruning.

Test 8 (7x6, 4, 6)
The game finishes in 16-18 moves averagely and MinMax algorithm evaluate the board 30.000times on average. For some moves, it wants to prevent me. Also it wants to win the game so it is not so successful for prevention. But if we look at the testes and compare them with the MinMax without pruning, we can see that there is indeed a huge.

5.4 Alpha-BetaPlayer vs. Alpha-BetaPlayer

This part, will be about the strength of the MinMax algorithm with pruning for both preventing and winning.

Test 1 (4x2, 2, 2)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 38 (for first player) and 13 (for second player) times. The first player wins. Unfortunately there is nothing much to do for the second player to prevent the first player winning. Also it can be seen that it performs less evaluation than MinMax without pruning.

Test 2 (4x2, 2, 6)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 37 (for first player) and 8 (for second player) times. It is strange that with more depth, we get less evaluation. I don't know why this happen.

Test 3 (4x2, 4, 2)
The game finishes in 7-8 moves averagely and MinMax algorithm evaluate the board 54 (for first player) and 41 (for second player) times. Unfortunately the second player does not make much to prevent the first player.

Test 4 (4x2, 4, 6)
The game finishes in 8 moves averagely and MinMax algorithm evaluate the board 435 (for first player) and 39 (for second player) times. The program has a fault on returning the column. It returned some invalid column numbers. I suspect that the problem can be in the Alpha-Beta algorithm, because it seems the second player makes evaluation too less. Bu I am not sure and the reason can be also in tree, because the MinMax traverse on the tree.

Test 5 (7x6, 2, 2)
The game finishes in 3 moves averagely and MinMax algorithm evaluate the board 144 (for first player) and 72 (for second player) times. Nothing unusual.

Test 6 (7x6, 2, 6)
The game finishes in 3-4 moves averagely and MinMax algorithm evaluate the board 536 for first player) and 97 (for second player) times. There are two remarkable points. The first one is indeed the pruning perfoms well. For this test case the first player

runs on MinMax without pruning did 36.000 evaluation. Also we can see that now, the complexity calculation is correct, with more depth there are more evaluation.

Test 7 (7x6, 4, 2)
The game finishes in 20-22 moves averagely and MinMax algorithm evaluate the board 841(for first player) and 649 (for second player) times. Surprisingly the second player wins.

Test 8 (7x6, 4, 6)
The game finishes in 18-20 moves averagely and MinMax algorithm evaluate the board 58.000 (for first player) and 47.000 (for second player) times. It behaves as I expected, the players want to fill the column up, again because of the heuristics, I suppose. This is also a clear sign that our pruning works correctly.

6 Discussion and Results

There are some remarkable results on two topics. The first one is the goal of this project, understand and compare the complexity difference of the MinMax algorithms without pruning and with Alpha-Beta pruning. The second one is again another goal of this project, create a good AI program which acts reasonable. In this section, I will discuss the results I have found from my tests about these topics.

6.1 Comparison of Complexities of MinMax Algorithms

We have calculated that MiniMax without pruning has the complexity of $\mathcal{O}(c^d)$ But this does not hold every time. We have seen that in some cases especially if goal number N is too low (i.e. 2) and if the depth is higher (i.e. 6) there will not be more evaluation than when the depth is lower. So we can say that since the game ends in early stages, the branching factor will be close to zero, so the evaluation number will not be higher. We can see this in Alpha-Beta pruning implementation also.

We can say that we have calculated the evaulation times for a player in a default game with depth 6, is 350.000. But if we look for the complexity of MinMax, the branching factor is 7 (since there are 7 playable moves at each point, maximumly) and the depth is 6. Then there must be $7^6 = 352.947$ at each move. It is near to the our result, but we get this result after 18-20 moves which is 9-10 moves for one player. So we should get 10 times more what we got. But I can not say it is wrong, because if we look we have calculated the upper bound of our algorithm with using "Big-O". But it is remarkable.

We can say that indeed pruning lower the complexity in every cases. If we can look for the same test cases done between MinMaxPlayer and Alpha-BetaPlayer, we see that it takes nearly half time for Alpha-BetaPlayer with pruning. Indeed it is amazing that such a small checks (alpha, beta checks) can make better the complexity of the algorithm.

6.2 Evaluation of AI

It can be seen that the AI program does not work so perfect. In some testes, it can be visible that it does not prevent the opponents win, even it is too obvious. For example in a moment, where there is three Xs in one row, and only one more is needed, the second player, O does not play to prevent X from winning. Maybe there must be better heuristics, but since it is not the main goal of the project, I didn't try to implement.

But also it is important to see that indeed AI works as it was programmed and makes rational moves when it can make. The first player indeed wants to win and the second player indeed wants to prevent the first player winning the game. But this optimization is highly related to the depth also.

As expected, even though the complexity of MinMax with pruning is improved, the AI is not improved, it stays same.

7 Appendix

In this section we will explain some alternative design choices. The most important one is adding the depth parameter for MinMax algorithm. I believe it will change nothing and it is not needed because at first we already build a tree with a maximum depth. But if we really want to make it in classical way, the base case of MinMax algorithm and the recursive calls will be changed, else will remain the same. The algorithm can be seen in Algorithm 4

Algorithm 4 Classical MinMax

```
 1: procedure MINMAX($node, depth$)
 2:     if depth ==0 or node.children.isEmpty() then  ▷ If node has no child then it is a leaf
 3:         return heuristic(node.board, node.player)    ▷ return heuristic evaluation
 4:     else
 5:         if node.children == MaximizingPlayer then
 6:             maximumValue ← $-\infty$
 7:             for every child c of node  do
 8:                 value← minMax(c, depth-1)
 9:                 maximumValue ← maxOf(value, maximumValue)
10:             end for
11:             return $maximumValue$
12:         else                                        ▷ It is minimizing player
13:             minimumValue ← $+\infty$
14:             for every child c of node  do
15:                 value← minMax(c, depth-1)
16:                 minimumValue ← minOf(value, minimumValue)
17:             end for
18:             return $minimumValue$
19:         end if
20:     end if
21: end procedure=0
```

```java
private int minMax(Node node, depth-1) {
    if (depth == 0||node.children.isEmpty())
    {
      return heuristic.evaluateBoard(node.player, node.board);
    }
...
    int minMaxvalue = minMax(child, depth-1);
...
}
```

Also the Alpha-Beta pruning function will be changed but for simplicity, I will not illustrate in here. The whole implementation can be found at repository :
gitlab/ege.sari/four-in-a-row