AI PRINCIPLES & TECHNIQUES

Four in a Row Assignment

Burak Balci s1073728
Radboud University 06/10/2023

Contents

1 Introduction

The "N in a Row" assignment is an implementation of a game called "Connect 4", a popular two-player board game where the objective is to be the first player to place a certain number of game pieces in a row, column, or diagonal on a game board. The specific objective of this project is to implement different algorithms/heuristics for the game.
The primary goals of this project are as follows:

- Implement the Tree structure to store game states.
- Implement AI algorithms such as MinMax algorithm and Alpha-Beta pruning.
- Experiment with the AI algorithms to make sure they are implemented correctly.
- Discuss the time complexities of both algorithms.
- **(Bonus)** Implement an improved heuristic.

2 Design

The code implementation for the "N in a Row" project consists of several key components, including the game board, player controllers, heuristic evaluation functions, and AI algorithms. In this section, we will provide an overview of these components and their functionalities.

2.1 App

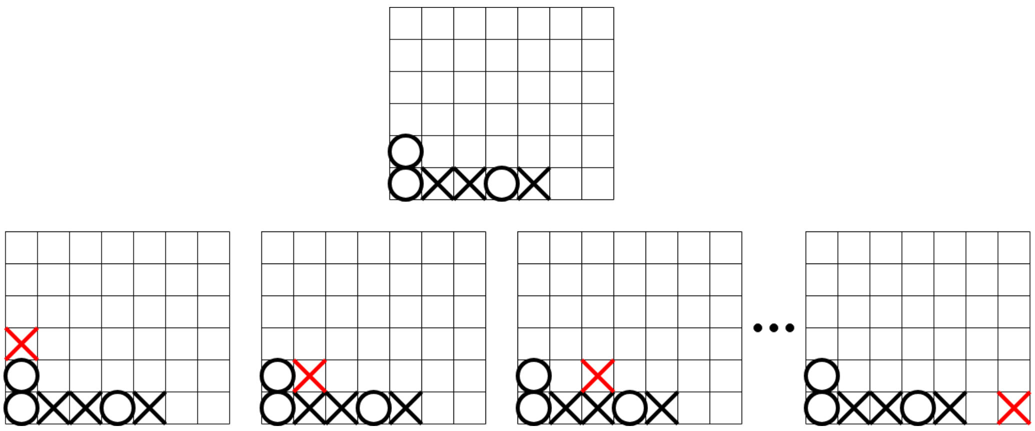
It is really a small thing and not relevant for the purpose of the assignment, but I made a few changes to the `App.java` class so that it asks user to select players(human, MinMax, Alpha-Beta) and depth. I just think that that way it is easier to test the code and it speeds up the process a little bit.

2.2 Game Board

The game board is represented as a two-dimensional grid where players take turns placing their pieces. The board class (`Board.java`) provides methods for checking the validity of moves, making moves, and determining the winner of the game. It also includes a method for displaying the current state of the board in a human-readable format.

2.3 Node and Tree Structure

The `Node` and `Tree` classes are central to AI player decision-making. The `Node` class represents a board state with a specific player's turn, while the `Tree` class constructs a game tree of potential moves and outcomes creating child(ren) node(s) with the given depth. A part of a tree can be seen below as example.



2.4 Player Controllers

Player controllers are responsible for making moves on the game board. There are two main types of player controllers implemented in the project:

2.4.1 HumanPlayer

This player controller allows human players to interact with the game through the console. It displays the current state of the board and prompts the player for their move. There is no algorithm behind it. Simply, user chooses which column they want to play and their piece(X or O) is played at that column.

2.4.2 AI Players (MinMaxPlayer and AlphaBetaPruning)

These player controllers implement AI-based algorithms for making optimal moves. They use the MinMax and Alpha-Beta Pruning algorithms to evaluate possible moves and choose the best one. Design of these algorithms and the explanation can be found at **section 2.6** in AI Algorithms.

2.5 Heuristic Evaluation

Heuristic evaluation functions (`Heuristic.java`) are used by AI players to assess the quality of a game state. The `SimpleHeuristic` class (`SimpleHeuristic.java`) provides a simple heuristic that calculates the utility of a game state based on the number of player pieces in a row, column, or diagonal. The heuristic functions are essential for AI players to make informed decisions. Unfortunately, I didn't add my own heuristic function and used the given heuristic function for this project.

2.6 AI Algorithms

The project includes two AI algorithms for AI players:

2.6.1 MinMax Algorithm

The MinMax algorithm (`MinMaxPlayer.java`) is implemented to perform a depth-limited search of the game tree. It evaluates potential moves by simulating future game states and selects the move that maximizes the player's chances of winning while considering the opponent's moves. So basically the maximizing player(X) wants to maximize it's advantage and minimizing player(O) wants to minimize maximizing players advantage. In the MinMax algorithm, the maximizing player explores its possible moves by recursively applying the algorithm to its children and ultimately selects the move that leads to the maximum value. On the other hand, the minimizing player follows a similar process, exploring its potential moves through MinMax algorithm recursively and choosing the move that results in the minimum value. If a node has no children, indicating the end of the game or reaching the maximum depth, it returns its heuristic value. For example, let's assume we are the maximizing player. That means we should select the node with maximum value from the child nodes of our current node. When assigning values to those child nodes, we are looking at child nodes of those each child nodes, which is children of a child node, and take the minimum value. We do that until we reach a node where game state is evaluated with a

```
function MinMax(node, depth, maximizingPlayer):
    if depth is 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer:
        value = MIN
        for each child of node:
            tempValue = MinMax(child, depth - 1, FALSE)
            value = max(value, tempValue)
        return value

    else:
        value = MAX
        for each child of node:
            tempValue = MinMax(child, depth - 1, TRUE)
            value = min(value, tempValue)
        return value
```

When I was implementing this algorithm, I didn't need `depth` and `maximizingPlayer` because class `Node` had everthing I needed. If `node.childNodes.isEmpty()` is true, that means depth is 0 since node with 0 depth will not have any child nodes and `node.getPlayerId()` gives the node ID so algorithm can understand if the current node is a node of maximizing player or not.

2.6.2 Alpha-Beta Pruning

The Alpha-Beta Pruning algorithm (`AlphaBetaPruning.java`) is an enhancement of the MinMax algorithm. It reduces the number of nodes evaluated by pruning branches of the game tree that are guaranteed to be suboptimal. This optimization improves the efficiency of AI players. We use two values named `alpha` and `beta`, for maximizing and minimizing players respectively. For the maximizing player, if the MinMax value of the child node is higher than the current value of the node, it is updated. Then if the current value of node is higher than alpha value, alpha is updated. Lastly we check if beta value is less than alpha value. If that's the case, then we prune rest of the child nodes/branches. If not, we continue just like the MinMax algorithm. Same happens for minimizing player. if the MinMax value of the child node is less than the current value of the node, it is updated. Then if the current value of node is less than beta value, beta is updated. Lastly we check if beta value is less than alpha value. If that's the case, then we prune rest of the child nodes/branches. If not, we continue just like the MinMax algorithm. Here is the pseudo code:

```
function Alpha−Beta(node, depth, alpha, beta, maximizingPlayer):
    if depth is 0 or node is a terminal node:
        return the heuristic value of node

    if maximizingPlayer:
        value = MIN
        for each child of node:
            tempValue = Alpha−Beta(child, depth − 1, alpha, beta, FALSE)
            value = max(value, tempValue)
            alpha = max(alpha, value)
            if beta <= alpha:
                break
        return value

    else:
        value = MAX
        for each child of node:
            tempValue = Alpha−Beta(child, depth − 1, alpha, beta, TRUE)
            value = min(value, tempValue)
            beta = min(beta, value)
            if beta <= alpha:
                break
        return value
```

Again, when I implemented the algorithm, I didn't need `depth`, and `maximizingPlayer` because `Node` class has those features in it.

3 Complexity

In this section we will discuss the complexity of MinMax algorithm and Alpha-Beta pruning. I included the pseudo codes for each algorithm again so we can look at the time complexity on the code. Since both algorithms use `SimpleHeuristic`, we will take $\mathcal{O}(H)$ as the time complexity of the heuristic function. However, since it's out of scope, we can assume it's constant.

3.1 MinMax Algorithm

We can think of MinMax algorithm as a recursive function that finds the best move by maximizing its possible moves and minimizing opponents possible moves. It does that kind of like Depth First Search, until depth is 0, since it evaluates boards with 0 depth, and find the MinMax value for bottom to top. So in the worst case, the MinMax algorithm must explore all possible game states to determine the optimal move. Therefore, the time complexity grows exponentially with the branching factor and the depth of the tree. Then we can say that the time complexity is $\mathcal{O}(c^d)$ where c is the branching factor, the number of possible moves at each decision point, in our game it's equal to `board.width` if every move is possible, and d is depth.

3.2 Alpha-Beta Pruning

Worst case time complexity is equal to MinMax algorithms time complexity, which is $\mathcal{O}(c^d)$. That's because having worst case means none of the child nodes has been pruned and we basically have the same algorithm. Best case occurs in the left side of the tree, which means it will go deep twice as MinMax algorithm in the same amount of time. Hence, time complexity will be the square root of MinMax algorithm, which is $\mathcal{O}(c^{\frac{d}{2}})$

3.3 Comparison

Alpha-Beta pruning improves the time complexity from $\mathcal{O}(c^d)$ to $\mathcal{O}(c^{\frac{d}{2}})$. It doesn't evaluate boards that are irrelevant in finding the optimal move for the user, so it is more efficient. Further testing and experiments will give some examples and more explanation in **Section 5**.

4 Implementation

In this section, implementation of classes, functions are explained. Documentation of the code will not be included here, however they are added to the actual code.

4.1 Node and Tree Structure

4.1.1 Node

Node structure uses three attributes. They are a board item which indicates the board state of that node, player id to check if current node should maximize or minimize for the AI algorithms and lastly a list of child nodes which are nodes with possible moves played on current nodes board. I didn't include depth here, because as I already explained, it is not needed since AI algorithms check if the current node has any child nodes which means depth is reached to 0. I also added getter functions for board, player id and child nodes, and a function to add a node to child nodes list. Here is the `Node.java` class I implemented:

```java
public class Node {
    public Board board;
    public int playerId;
    public List<Node> childNodes;

    public Node(Board board, int playerId) {
        this.board = board;
        this.playerId = playerId;
        this.childNodes = new ArrayList<>();
    }

    public Board getBoard() {
        return board;
    }

    public int getPlayerId() {
        return playerId;
    }

    public List<Node> getChildNodes() {
        return childNodes;
    }

    public void addChildNode(Node child) {
        this.childNodes.add(child);
    }
}
```

4.1.2 Tree

Tree structure uses `buildTree` function, which returns a `Node` to build a tree. It needs board, player id, depth and game N which is how many pieces needed horizontally, vertically or diagonally to win the game. There is recursive function `buildTree`, which creates a node with the given board state and player id, then if that node is not an already won game or depth is not zero(higher), it copies the board for every possible moves and checks if that move is playable. If it is, plays that move to the copy of the board, and calls `buildTree` again for that child node. Finally, adds the child node to the child node list of the parent node. That way, tree is created. Function returns the starting node(`root`). I also added a get function that returns the root of tree. Here is the `Tree.java` class I implemented:

```java
public class Tree {
    public Node root;

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

    public Node buildTree(Board board, int playerId, int depth, int gameN)
     {
        Node node = new Node(board, playerId);
        int winning = Game.winning(board.getBoardState(), gameN);
        if (winning == 0 && depth > 0) {
            for (int i = 0; i < board.width; i++) {
                Board boardCopy = new Board(board);
                if (boardCopy.play(i, playerId)) {
                    int nextPlayerId = 3 - playerId;
                    Node child = buildTree(boardCopy, nextPlayerId, (depth
                         - 1), gameN);
                    node.addChildNode(child);
                }
            }
        }
        return node;
    }

    public Node getRoot() {
        return root;
    }
}
```

4.2 Implementation of AI Algorithms

A simple explanation of implementation was already given in sections **2.6.1** and **2.6.2**. In this section, we will dig deeper and look also at the other functions used in `MinMaxPlayer.java` and `AlphaBetaPruning.java`. Function `makeMove` is almost the same for both classes, so I will explain the logic only once. It mostly applies for both with a small difference.

4.2.1 Make Move

The main idea behind it is that to print the best column to play using an AI algorithm. Function copies the current board for each possible move and creates a new tree for that board. Then it finds the best option among these trees by using the MinMax algorithm or Alpha-Beta pruning. It prints the recommended column, and lastly, user chooses the column they want to play. The difference between `makeMove` for MinMax player and Alpha-Beta pruning is Alpha-Beta pruning also uses alpha and beta values when finding the value of a possible move. Implementation can be found at the end of **section 4.2.1** and **section 4.2.2**.

4.2.2 MinMax Player

MinMax player uses four attributes. These are player id, game N, depth and heuristic which is used to evaluate a board state. Depth is only being used for making a move, but not for the main MinMax algorithm. Class contains two functions `makeMove` and `minMaxMove`. Since `makeMove` function is already explained in the **previous section**, I will only explain the implementation of `minMaxMove` here.
In `minMaxMove`, the goal is to find the MinMax value of a board. The implementation is really similar to the pseudo code given in **section 2.6.1**, however as I said earlier, I didn't use depth as a parameter. If a node does not have any child nodes, which means either game is over or depth is reached, it returns the heuristic value of the board using `evaluateBoard` function. If a node does have child nodes, then algorithm checks if that node belongs to a maximizing player or a minimizing player by using `getPlayerId` function. If it is a maximizing player, then MinMax algorithm is recursively applied for each child node, and returns max value. If it is a minimizing player, then again MinMax algorithm is recursively applied for each child node, but this time returns min value. Here is the implementation of `MinMaxPlayer.java`:

```java
public class MinMaxPlayer extends PlayerController {
    private final int depth;
    Scanner scanner = new Scanner(System.in);
    final int MIN = Integer.MIN_VALUE;
    final int MAX = Integer.MAX_VALUE;
    final int maxPlayerId = 1;

    public MinMaxPlayer(int playerId, int gameN, int depth, Heuristic
        heuristic) {
        super(playerId, gameN, heuristic);
        this.depth = depth;
    }

    @Override
    public int makeMove(Board board) {
        System.out.println(board);
        if (heuristic != null) {
            int move = 0;
            int maxValue = MIN;
            for (int i = 0; i < board.width; i++) {
                if (board.isValid(i)) {
                    Board boardCopy = board.getNewBoard(i, playerId);
                    Tree tree = new Tree(boardCopy, playerId, depth, gameN
                        );
                    int maxMove = minMaxMove(tree.getRoot());
                    if (maxMove > maxValue) {
                        maxValue = maxMove;
                        move = i;
                    }
                }
            }
            System.out.println("Heuristic: " + heuristic + " calculated
                the best move is: "
                    + (move + 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;
    }

    private int minMaxMove(Node node) {
        if (node.childNodes.isEmpty()) {
            return heuristic.evaluateBoard(node.getPlayerId(), node.
                getBoard());
        }
        else if (node.getPlayerId() == maxPlayerId) {
            int val = MIN;
            for (Node child : node.getChildNodes()) {
                int tempValue = minMaxMove(child);
                val = Math.max(tempValue, val);
            }
            return val;
        }
        else {
            int val = MAX;
            for (Node child : node.getChildNodes()) {
                int tempValue = minMaxMove(child);
                val = Math.min(tempValue, val);
            }
            return val;
        }
    }
}
```

4.2.3 Alpha-Beta Pruning

Alpha-Beta pruning uses six attributes. These are player id, game N, depth, heuristic, alpha and beta values. Depth is only being used for making a move, but not for the main Alpha-Beta algorithm. Class contains two functions `makeMove` and `alphaBetaMove`. Since `makeMove` function is already explained in the **previous section**, I will only explain the implementation of `alphaBetaMove` here.
In `alphaBetaMove`, the goal is to find the MinMax value of a board again, but by using Alpha-Beta pruning. The implementation is really similar to the pseudo code given in **section 2.6.2**, however as I said earlier, I didn't use depth as a parameter. If a node does not have any child nodes, which means either game is over or depth is reached, it returns the heuristic value of the board using `evaluateBoard` function. If a node does have child nodes, then algorithm checks if that node belongs to a maximizing player or a minimizing player by using `getPlayerId` function. If it is a maximizing player, then AlphaBeta pruning is recursively applied for each child node, and returns max value. If it is a minimizing player, then again AlphaBeta pruning is recursively applied for each child node, but this time returns min value. But while doing that, it prunes branches that are not necessary in finding the value of a node by comparing alpha and beta values. For maximizing player, it assigns the maximum value between value found and alpha value to alpha, and checks if alpha is higher than beta. For minimizing player, it assigns the minimum value between value found and beta value to beta, and checks if alpha is higher than beta. Here is the implementation of `AlphaBetaPruning.java`:

```java
public class AlphaBetaPruning extends PlayerController {
    private final int depth;
    Scanner scanner = new Scanner(System.in);
    final int MIN = Integer.MIN_VALUE;
    final int MAX = Integer.MAX_VALUE;
    final int Alpha = Integer.MIN_VALUE;
    final int Beta = Integer.MAX_VALUE;
    final int maxPlayerId = 1;

    public AlphaBetaPruning(int playerId, int gameN, int depth, Heuristic
        heuristic) {
        super(playerId, gameN, heuristic);
        this.depth = depth;
    }

    @Override
    public int makeMove(Board board) {
        System.out.println(board);
        if (heuristic != null) {
            int move = 0;
            int maxValue = MIN;
            for (int i = 0; i < board.width; i++) {
                if (board.isValid(i)) {
                    Board boardCopy = board.getNewBoard(i, playerId);
                    Tree tree = new Tree(boardCopy, playerId, depth, gameN
                        );
                    int maxMove = alphaBetaMove(tree.getRoot(), Alpha,
                        Beta);
                    if (maxMove > maxValue) {
                        maxValue = maxMove;
                        move = i;
                    }
                }
            }
            System.out.println("Heuristic: " + heuristic + " calculated
                the best move is: "
                    + (move + 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;
    }

    private int alphaBetaMove(Node node, int a, int b) {
        if (node.childNodes.isEmpty()) {
            return heuristic.evaluateBoard(node.getPlayerId(), node.
                getBoard());
        }
        else if (node.getPlayerId() == maxPlayerId) {
            int val = MIN;
            for (Node child : node.getChildNodes()) {
                int tempValue = alphaBetaMove(child, a, b);
                val = Math.max(tempValue, val);
                a = Math.max(val, a);
                if (b <= a) {
                    break;
                }
            }
            return val;
        }
        else {
            int val = MAX;
            for (Node child : node.getChildNodes()) {
                int tempValue = alphaBetaMove(child, a, b);
                val = Math.min(tempValue, val);
                b = Math.min(val, b);
                if (b <= a) {
                    break;
                }
            }
            return val;
            }
        }
}
```

5 Tests

In this section, we describe the experiments conducted using the "N in a Row" code implementation and present the results obtained from these experiments. The primary goal of these experiments is to evaluate the performance of different AI algorithms. I decided to use two board sizes, two game N's and two depths. I don't think running tests with different depths when game N is 2 is necessary, since game will probably end in 3 moves. So I will eliminate tests with game N is 2 and depth is 4. That means there will be 6 tests for each experiment. And total of 18 tests since I'm using three possible mathups.

- Matchups (Human vs. MinMax, Human vs. AlphaBeta, MinMax vs. AlphaBeta)
- Board sizes (5x3, 7x6)
- Different gameN's (2, 4)
- Different depths (2, 4)

I except Human vs. AI algorithm tests will result in mostly wins for Human(me) or maybe draws. I don't except AI to win a lot because the heuristic function is not that good. I didn't think there will be any difference between AI vs. AI tests result wise, since at the end they both use a version of MinMax algorithm. That's why I will not test MinMax vs. MinMax or AlphaBeta vs. AlphaBeta. But instead, I will test MinMax vs. AlphaBeta, so it is easier to compare each algorithm head to head.
Each experiment format is like board, gameN, depth. For example, (2x2, 1, 2) means board size is 2x2, game N is 1 and depth is 2.

5.1 Experiment 1: Human vs. MinMaxPlayer

In this experiment, I will play against MinMaxPlayer a few times. My expectation is that I win most of the games, but when game N is 2 and MinMaxPlayer starts, I might not be able to prevent MinMaxPlayer from winning.

Experiments:

- **(5x3, 2, 2):**
  In every test I ran, game ended after 3 moves with winner being whichever player who started. MinMaxPlayer evaluates the board 70-120 times depending on my moves.
- **(5x3, 4, 2):**
  This time, some of the games ended in a tie. I think that's because making 4 in a row in such small board is a little hard. Especially when making a diagonal 4 in a row is impossible. But still, I won most of the games. MinMaxPlayer evaluates the board 350-420 times depending on my moves.
- **(5x3, 4, 4):**
  Again similarly, some of the games ended in a tie, this time algorithm actually blocked some of my winning moves. I still won most of the games. It took longer for me to win when MinMaxPlayer started. MinMaxPlayer evaluates the board 6000-6500 times depending on my moves.

- **(7x6, 2, 2):**
  Since game N is 2, game ended in 3 moves every time. Again with starting player winning. MinMaxPlayer evaluates the board 220-400 times depending on my moves.

- **(7x6, 4, 2):**
  I ran a few tests, and even when I don't play a good move, algorithm still can't find a way to win. I assume that's because depth is low, and the goal is high. MinMaxPlayer evaluates the board 1000-1700 times depending on my moves.

- **(7x6, 4, 4):**
  When I started, games ended mostly in 7-9 moves since algorithm usually doesn't block my winning moves. When MinMaxPlayer started, I tried blocking it moves so games took longer. But I still won every game. MinMaxPlayer evaluates the board 45000-100000 times depending on my moves. The reason for such a big gap is when I let MinMaxPlayer start, I tried blocking its moves a lot, so it evaluated way more compared to me starting and finishing the game in 7-9 moves.

To sum up, MinMaxPlayer doesn't block my winning moves mostly, so unless it's a game with N is 2, which means when MinMaxPlayer starts, it wins in its next move, or a 5x3 with N is 4, which is hard to win since diagonal 4 in a row is impossible, I win mostly quickly depending on the board size. Further analysis can be found in **Section 6**.

5.2 Experiment 2: Human vs. AlphaBetaPruning

Again, I will play against AlphaBetaPruning a few times. I expect similar results like Experiment 1. However, this time I think AlphaBetaPruning will evaluate the board less compared to MinMaxPlayer because it prunes branches. Since it will have similar results, experiments below will have really similar text to Experiment 1, but I will update how many times the algorithm evaluates the board.

Experiments:

- **(5x3, 2, 2):**
  In every test I ran, game ended after 3 moves with winner being whichever player who started. AlphaBetaPruning evaluates the board 20-70 times depending on my moves.

- **(5x3, 4, 2):**
  This time, some of the games ended in a tie. I think that's because making 4 in a row in such small board is a little hard. Especially when making a diagonal 4 in a row is impossible. But still, I won most of the games. AlphaBetaPruning evaluates the board 160-200 times depending on my moves.

- **(5x3, 4, 4):**
  Again similarly, some of the games ended in a tie, this time algorithm actually blocked some of my winning moves. I still won most of the games. It took longer for me to win when AlphaBetaPruning started. AlphaBetaPruning evaluates the board 600-1000 times depending on my moves.

- **(7x6, 2, 2):**
  Since game N is 2, game ended in 3 moves every time. Again with starting player winning. AlphaBetaPruning evaluates the board 70-150 times depending on my moves.

- **(7x6, 4, 2):**
  I ran a few tests, and even when I don't play a good move, algorithm still can't find a way to win. I assume that's because depth is low, and the goal is high. AlphaBetaPruning evaluates the board 350-580 times depending on my moves.

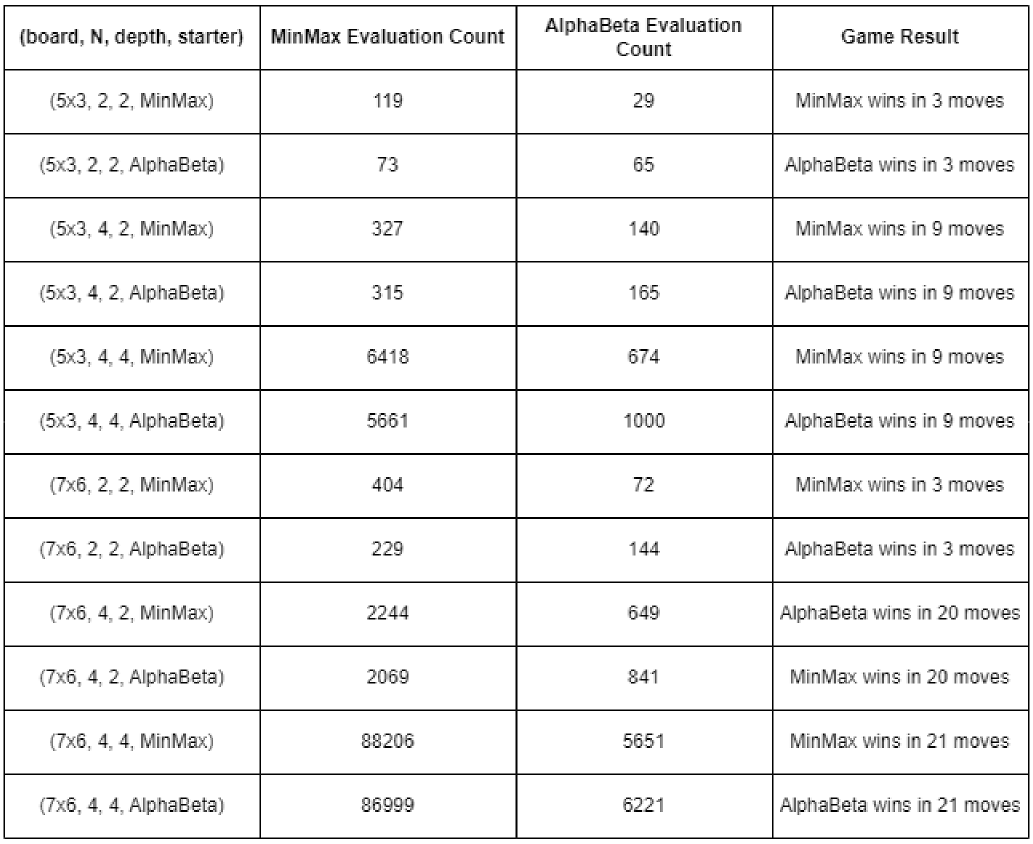
- **(7x6, 4, 4):**
  When I started, games ended mostly in 7-9 moves since algorithm usually doesn't block my winning moves. When AlphaBetaPruning started, I tried blocking it moves so games took longer. But I still won every game. AlphaBetaPruning evaluates the board 3000-5000 times depending on my moves. The reason for such a big gap is when I let AlphaBetaPruning start, I tried blocking its moves a lot, so it evaluated way more compared to me starting and finishing the game in 7-9 moves.

To summarize this experiment, game results were similar to Experiment 1. But there were significant change in evaluation count. In the next experiment, we will actually compare those algorithms head to head and see the difference in evaluation counts. Further analysis can be found in **Section 6**.

5.3 Experiment 3: MinMaxPlayer vs. AlphaBetaPruning

For this experiment, I will create a table for each test I run. Running two tests for each case is smart, one starting with MinMaxPlayer, one starting with AlphaBetaPruning. That's because for a certain case, starting move for an algorithm will always be the case, there won't be much variations like Human vs. AI experiments.

Experiments:



(board, N, depth, starter)	MinMax Evaluation Count	AlphaBeta Evaluation Count	Game Result
(5x3, 2, 2, MinMax)	119	29	MinMax wins in 3 moves
(5x3, 2, 2, AlphaBeta)	73	65	AlphaBeta wins in 3 moves
(5x3, 4, 2, MinMax)	327	140	MinMax wins in 9 moves
(5x3, 4, 2, AlphaBeta)	315	165	AlphaBeta wins in 9 moves
(5x3, 4, 4, MinMax)	6418	674	MinMax wins in 9 moves
(5x3, 4, 4, AlphaBeta)	5661	1000	AlphaBeta wins in 9 moves
(7x6, 2, 2, MinMax)	404	72	MinMax wins in 3 moves
(7x6, 2, 2, AlphaBeta)	229	144	AlphaBeta wins in 3 moves
(7x6, 4, 2, MinMax)	2244	649	AlphaBeta wins in 20 moves
(7x6, 4, 2, AlphaBeta)	2069	841	MinMax wins in 20 moves
(7x6, 4, 4, MinMax)	88206	5651	MinMax wins in 21 moves
(7x6, 4, 4, AlphaBeta)	86999	6221	AlphaBeta wins in 21 moves

From what I see, it's obvious that AlphaBetaPruning evaluates the board less than MinMaxPlayer, which is expected. And the moves they make are always the same, which is also expected. It seems like I was a little bit off with evaluation counts in first two experiments, but since the heuristic function I'm using is not that good, games can take longer, which explains the difference between the evaluation counts in Human vs. AI and AI vs. AI. Further analysis can be found in **Section 6**.

6 Results from the Experiments

The tests I ran provided valuable insights into the performance of different AI player algorithms. The following discussions and conclusions can be drawn from the results:

- The depth of the search tree affects AI player performance. Deeper searches lead to more competitive gameplay. Even though I kept the depth really close, which were 2 and 4, I felt like AI algorithm made a little smarter decisions when depth was 4. I think if I tested with a higher depth, the difference would've been higher. But still, if we look at MinMaxPlayer vs. AlphaBetaPruning experiment, when board is 7x6, game N is 4, when depth is 2, starting player loses the game. However, when depth is 4, starting player wins the game. This alone shows that deeper search makes the algorithm make better moves.
- The Alpha-Beta pruning is more efficient compared to MinMax algorithm. Since they are both technically a MinMax algorithm, result was always the same for both of these. But if we look at evaluation counts, we can see that AlphaBetaPruning evaluated the board way less compared to MinMaxPlayer despite always finding the same result.

- Heuristic evaluation function play a crucial role in AI players decision making. I assume more complex heuristic can actually lead to drawn games or even AI winning. However, in this case heuristic function was really bad so AI's got beaten easily. When testing, algorithms made really dumb moves. They didn't miss their winning moves, but they also didn't block opponents winning moves.

7 Conclusion

In this project, we developed an implementation of the classic board game N in a Row. We created a flexible and extensible system that allows different players, different board size, and different depth for AI algorithms. The AI players use the MinMax algorithm with/without Alpha-Beta pruning to evaluate possible moves and make a smart decision. Unfortunately, I couldn't implement a better heuristic due to lack of time, and that was the main reason that AI algorithms were easy to beat.
Through experimentation, we observed that the AI players with higher depth performed better. It was no surprise, since evaluation the game deeper helps making a smarter decision. Also it was obvious that pruning lowers the complexity of the algorithm.
Overall, this project highlights the application of AI algorithms and heuristics in board games. The system's modular design allows for easy extension and experimentation with different AI algorithms and heuristics, making it a valuable platform for future research and development in the field of board game AI.