# AI Principles and Techniques: Four in a Row Assignment

Tim Geristen s1041423 ,Vamsi Yerramsetti s1032599

November 15, 2021

## 1 Introduction and Project Description

Four in a row or "Connect 4" is a popular game which when played against a computer can be optimised with recursive search algorithms such as MiniMax. MiniMax produces the most optimal move for a player assuming that the player is playing to win. MiniMax is used in other games like chess,checkers etc . We are interested in implementing MiniMax and studying it's performance with and without alpha-beta pruning whose objective is to reduce the number of nodes searched. We showcase our tree structure we built to keep account of the game state and report the different search strategies and design choices we used. All code,images and tables and figures in this report were developed by the authors.

### 1.1 Hypothesis

Before we started this project we were advised by our TA to make a hypothesis on the effect of pruning. We believe and predict that by using alpha-beta pruning MiniMax will have a much faster run time and and better run time complexity. We feel the impact of alpha-beta pruning will grow exponentially as the depth increases.

## 2 MiniMax and Alpha beta pruning

### 2.1 MiniMax

MiniMax works on a tree of nodes, where each node is a possible scenario in the game, or as we call it "a game state". So in the beginning we have "N" possible moves(N is number of columns). We can imagine this as one node branched out to N nodes that each represent the opening move by player 1. MiniMax works by searching future potential moves and selecting the best most optimal move. In our case, at every turn we are trying to maximise our chances of winning whereas the computer is trying to minimize our chance of winning.Therefore by recursive depth search in possible game play scenarios the computer tries to win the game. [FK88]

### 2.2 Alpha beta pruning

Although MiniMax is certain to produce the best move at any given scenario , it requires excessive and unnecessary searching of branches which will not affect the outcome.In problems that use complex trees with a voluminous amount of branches and search directions , alpha beta pruning is very useful. Alpha-beta pruning prunes branches whose value will not affect the outcome by checking if the alpha(the highest value we found across the "maximiser") is greater than or equal to beta (the lowest values we found across the "minimizer"). If this condition holds we prune thus reducing the number of branches/nodes we search hence reducing the time spent to find the optimal move but most importantly improving the complexity of our strategy.[FK88]

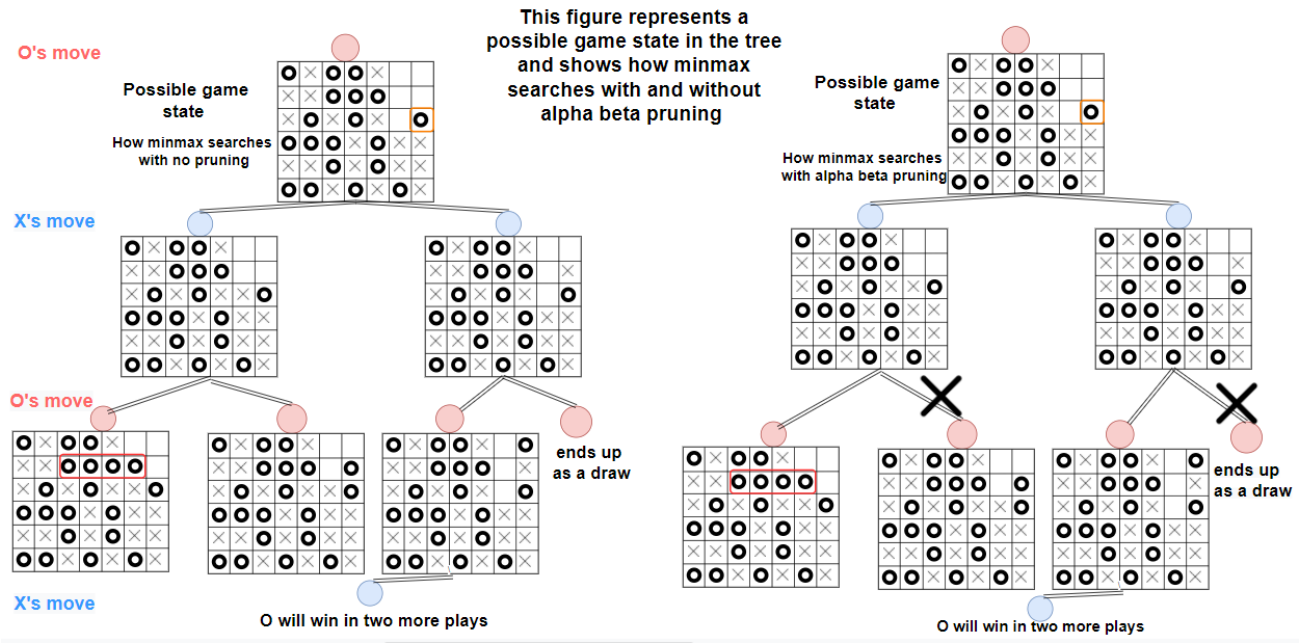



Figure 1: Possible game states represented in the tree

# 3 Implementation and Methodology

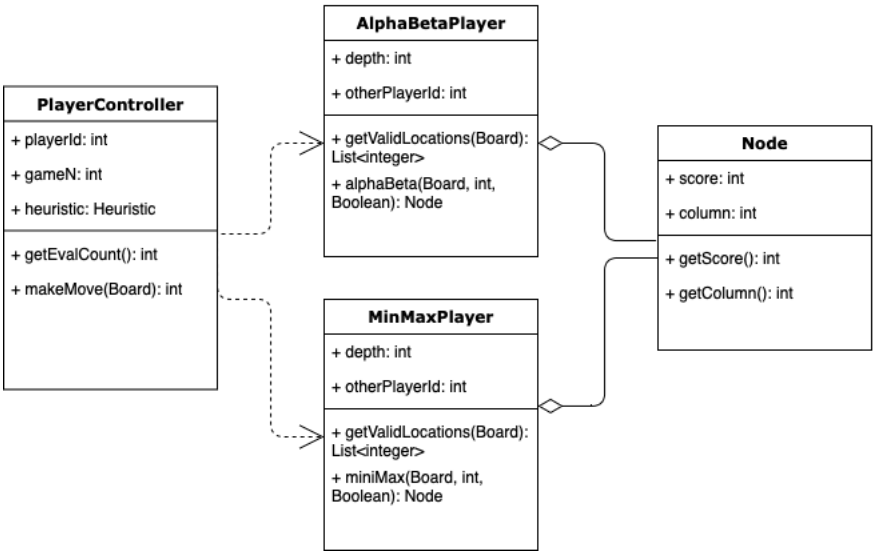
## 3.1 Design features and embedded tree implementation

We took a very straight-forward approach and embedded the tree structure within the minimax and alpha-beta method. We created a Node class for the tree structure where every Node object represents a different state of the board, containing 2 variables:

1. Score: this contains the score that board state gets by the Heuristic.
2. Column: the column played in that board-state, this is stored to be able to track back which move to make whenever the algorithm is done.

Whenever a state is explored, we return a new Node object in which we store both of these values. When the algorithm terminates it has retrieved the root node of a tree of which one of the leaf nodes had the highest score. Then the move is made in the stored column of that node.

The image below shows the Class Layout used in the code. The implementation of both the miniMax and alpha-beta algorithm relies on this layout.[DDP22]




Then there are 2 separate classes which both implement the Player Controller class, one which implements the Minimax algorithm and one which implements the Minimax algorithm with alpha-beta pruning.

The main algorithms are located in the two methods:

- miniMax(Board, int, Boolean) : Node
- alphaBeta(Board, int, int, int, Boolean) : Node

## 3.2 Implementation of MiniMax

The pseudo code for the miniMax method is as follows:

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value :=
        for each child of node do
            value := max(value, minimax(child, depth      1, FALSE))
        return value
    else (* minimizing player *)
        value := +
        for each child of node do
            value := min(value, minimax(child, depth      1, TRUE))
        return value
```

Our implementation for this method is as follows:

```
public Node miniMax(Board board, int depth, Boolean isMaxPlayer){
   //List of all valid locations the player is allowed to play in
   List<Integer> validLocations = getValidLocations(board);
   //Is this board in a winning or losing state?
   int isTerminalNode = Game.winning(board.getBoardState(), gameN);
   if(isTerminalNode != 0){
       //The MinMax Player won
       if(isTerminalNode == playerId){
           return new Node(Integer.MAX_VALUE, -1);
       }
       //The other player won
       else if(isTerminalNode == otherPlayerId){
           return new Node(Integer.MIN_VALUE, -1);
       }
       //It's a draw
```

```
        else if(isTerminalNode == -1){
            return new Node(0, -1);
        }
    }
    //Base case for recursive function, max depth reached
    else if (depth == 0){
        //Use the heuristic to evaluate the reached board state
        return new Node(heuristic.evaluateBoard(isMaxPlayer ? playerId : otherPlayerId,
             board), -1);
    }
    //Iterate through all possible board options with respect to the max depth
    int value;
    //The column to play in
    int column = new Random().nextInt(validLocations.size());
    if(isMaxPlayer) //Maximizing player
    {
        value = Integer.MIN_VALUE;
        //Iterate through all valid locations
        for(int i = 0; i < validLocations.size(); i++){
            int col = validLocations.get(i);
            Board copy = board.getNewBoard(col, playerId);
            //Recursive call
            int newScore = miniMax(copy, depth-1, false).getScore();
            if(newScore > value){
                value = newScore;
                column = col;
            }
        }
    }
    else //Minimizing player
    {
        value = Integer.MAX_VALUE;
        //Iterate through all valid locations
        for(int i = 0; i < validLocations.size(); i++){
            int col = validLocations.get(i);
            Board copy = board.getNewBoard(col, otherPlayerId);
            //Recursive call
            int newScore = miniMax(copy, depth-1, true).getScore();
            if(newScore < value){
                value = newScore;
                column = col;
            }
        }
    }
    return new Node(value, column);
}
```

### 3.3 Implementation of alpha-beta

The pseudo code for the alpha-beta pruning algorithm is as follows:

```
function alphabeta(node, depth, alpha, beta, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value :=
        for each child of node do
            value := max(value, alphabeta(child, depth     1,    ,    , FALSE))
            if value        then
                break (*    cutoff *)
            := max(   , value)
        return value
    else
        value := +
        for each child of node do
            value := min(value, alphabeta(child, depth     1,    ,    , TRUE))
            if value        then
                break (*    cutoff *)
            := min(   , value)
        return value
```

Our implementation for this method is as follows:

```
public Node alphaBeta(Board board, int depth, int alpha, int beta, Boolean isMaxPlayer
    ){
   //List of all valid locations the player is allowed to play in
   List<Integer> validLocations = getValidLocations(board);
   //Is this board in a winning or losing state?
   int isTerminalNode = Game.winning(board.getBoardState(), gameN);
   if(isTerminalNode != 0){
       //The MinMax Player won
       if(isTerminalNode == playerId){
           return new Node(Integer.MAX_VALUE, -1);
       }
       //The other player won
       else if(isTerminalNode == otherPlayerId){
           return new Node(Integer.MIN_VALUE, -1);
       }
       //It's a draw
       else if(isTerminalNode == -1){
           return new Node(0, -1);
       }
   }
   //Base case for recursive function, max depth reached
   else if (depth == 0){
       //Use the heuristic to evaluate the reached board state
       return new Node(heuristic.evaluateBoard(isMaxPlayer ? playerId : otherPlayerId,
            board), -1);
   }
   //Iterate through all possible board options with respect to the max depth
   int value;
   int column = new Random().nextInt(validLocations.size());
   if(isMaxPlayer) //Maximizing player
   {
       value = Integer.MIN_VALUE;
       for(int i = 0; i < validLocations.size(); i++){
           int col = validLocations.get(i);
           Board copy = board.getNewBoard(col, playerId);
           int newScore = alphaBeta(copy, depth-1, alpha, beta, false).getScore();
           if(newScore > value){
               value = newScore;
               column = col;
           }
           alpha = Integer.max(alpha, value);
           if(alpha >= beta){
               break;
           }
       }
   }
   else //Minimizing player
   {
       value = Integer.MAX_VALUE;
       for(int i = 0; i < validLocations.size(); i++){
           int col = validLocations.get(i);
           Board copy = board.getNewBoard(col, otherPlayerId);
           int newScore = alphaBeta(copy, depth-1, alpha, beta, true).getScore();
           if(newScore < value){
               value = newScore;
               column = col;
           }
           beta = Integer.min(beta, value);
           if(alpha >= beta){
               break;
           }
       }
   }
   return new Node(value, column);
}
```

# 4 Testing and Experiments

We measure the the complexity of minimax with and without alpha-beta pruning using the run time taken.[Fri14] We focus on what affects the run time and start our testing and experimenting from there. We do the final evaluation of a game at leaf nodes. Hence this along with our heuristic can be taken as a standard to measure the run time.

We experimented with different board sizes,tree depths,and different values of N which is the number of X's or O's a player has to get to win. So in total we have three main attributes that affect the run time and we are interested in testing out the magnitude of their influence to the overall run time. We can do this by declaring two of them to test the third. This way of selective testing will provide us with a greater insight to each attribute's influence.[THA17]

The combinations we did are :

1. Size of board and N $->$ Search Depth
2. Depth and N $->$ Size of Board
3. Depth and Size of board $->$ N

| **N/Size of the Board** | | **4x5** | **6x7** | **8x9** | **10x11** |
|---|---|---|---|---|---|
| 3 | Nr of MiniMaxPlayer evaluations | 1829 | 9665 | 34049 | 92373 |
| | Nr of AlphaBetaPlayer evaluations | 118 | 267 | 551 | 966 |
| | **Ratio** | **15.5** | **36.2** | **61.8** | **95.6** |
| 4 | Nr of MiniMaxPlayer evaluations | 2026 | 15360 | 61672 | 185176 |
| | Nr of AlphaBetaPlayer evaluations | 185 | 826 | 1889 | 3550 |
| | **Ratio** | **10.9** | **18.6** | **32.6** | **52.1** |
| 5 | Nr of MiniMaxPlayer evaluations | 2061 | 16184 | 66930 | 211281 |
| | Nr of AlphaBetaPlayer evaluations | 196 | 921 | 2375 | 5134 |
| | **Ratio** | **10.5** | **17.6** | **28.1** | **41.2** |

Table 1: $N$ and board size results $\Rightarrow searchdepth = 4$

| **N/Search Depth** | | **3** | **4** | **5** | **6** |
|---|---|---|---|---|---|
| 3 | Nr of MiniMaxPlayer evaluations | 410 | 1725 | 7599 | 35437 |
| | Nr of AlphaBetaPlayer evaluations | 86 | 109 | 348 | 625 |
| | **Ratio** | **4.8** | **15.9** | **21.8** | **56.7** |
| 4 | Nr of MiniMaxPlayer evaluations | 448 | 2026 | 9707 | 38764 |
| | Nr of AlphaBetaPlayer evaluations | 167 | 185 | 1063 | 1040 |
| | **Ratio** | **2.7** | **10.9** | **9.1** | **37.3** |
| 5 | Nr of MiniMaxPlayer evaluations | 524 | 2061 | 8363 | 41521 |
| | Nr of AlphaBetaPlayer evaluations | 193 | 196 | 990 | 2564 |
| | **Ratio** | **2.7** | **10.5** | **8.44** | **16.2** |

Table 2: $N$ and search depth $\Rightarrow boardsize = 4x5$

| Search Depth/Size of the Board | | 5x4 | 7x6 | 9x8 | 11x10 |
|---|---|---|---|---|---|
| 3 | Nr of MiniMaxPlayer evaluations | 410 | 1228 | 2690 | 6062 |
| | Nr of AlphaBetaPlayer evaluations | 86 | 177 | 287 | 418 |
| | **Ratio** | **4.8** | **6.9** | **9.4** | **14.5** |
| 4 | Nr of MiniMaxPlayer evaluations | 1725 | 9665 | 34049 | 92373 |
| | Nr of AlphaBetaPlayer evaluations | 109 | 267 | 551 | 966 |
| | **Ratio** | **15.8** | **36.2** | **61.8** | **95.6** |
| 5 | Nr of MiniMaxPlayer evaluations | 7559 | 83716 | 380222 | 1032353 |
| | Nr of AlphaBetaPlayer evaluations | 348 | 2130 | 5344 | 8303 |
| | **Ratio** | **21.8** | **39.3** | **71.1** | **124.3** |
| 6 | Nr of MiniMaxPlayer evaluations | 35581 | 455002 | 2753618 | 9208191 |
| | Nr of AlphaBetaPlayer evaluations | 625 | 2502 | 6866 | 10350 |
| | **Ratio** | **56.9** | **181.6** | **401.1** | **889.7** |

Table 3: Search depth and board size ⇒N= 3

# 5 Complexity Analysis

As long as we keep the board width finite the game "Four in a Row" is in P as there is a finite size and all possible game states/positions can be looked up in a database. However deciding if a player wins is a PSPACE complete problem due the polynomial amount of moves.We believe that this game can be imagined as a SAT problem where we can note all possible game states adding repetitive clauses and variables which creates a network of instances the SAT solver can search.In the end our clauses should identify a win from which we can induce a truth variable.So to tackle such a game we must go in depth of the complexity of its underlying search algorithms.

## 5.1 Complexity of MiniMax

MiniMax performs a depth-first search on the tree.Hence its time complexity is determined by the tree's branching factor(number of children per node on average) and its depth. In the MiniMax function we recursively call the function *validlocation.size*() times for both minimizing and maximizing plays. Every time the function is recursively called the depth argument decrements till its one.[**?**]

```
for(int i = 0; i < validLocations.size(); i++){    //O(valid.location.size)
    int col = validLocations.get(i);
    Board copy = board.getNewBoard(col, playerId);
    int newScore = miniMax(copy, depth-1, false).getScore();
```

So in terms of time complexity:

- $T = O(validlocation.size^n)$
- In the worst case, *validlocation.size* = width of the board.
- So the worst case time complexity of MiniMax = $O(w^d)$ where w=width of the board and d is the depth.Hence is is exponential.
- The best case time complexity is the same as the worst case ,its run-time depends on the values of "d" and "w".
- The worst case space complexity will then be the width of the board(as that's number of legal moves a player has at max) times the depth so $O(wd)$

## 5.2 Complexity of MiniMax with alpha beta pruning

Alpha Beta pruning prune leafs/sub-trees that satisfy the condition of alpha ¿= beta.This is a big advantage when we work with trees with large branching factors and depth.Ex:Chess,Connect4

Just like in the MiniMax function the alpha-beta function is recursively called once for both minimizing and maximizing plays.It has the same for-loop and decrements the depth at each call.However this function has an extra operation. It will prune the search path if at any given point where a node's

alpha value is greater or equal to its beta value.In terms of complexity this has a large effect as we are reducing possible search paths by pruning the sub-trees/leafs.[THA17]

```
int newScore = alphaBeta(copy, depth-1, alpha, beta, false).getScore();
if(newScore > value){
    value = newScore;
    column = col;
}
alpha = Integer.max(alpha, value);
if(alpha >= beta){
    break;
```

However, in the worst case scenario nothing gets pruned and hence we just did a normal MiniMax DFS.So the worst case time complexity of MiniMax with alpha beta pruning is $O(w^d)$. However in the best case scenario alpha beta pruning takes place whenever possible.This means both players play to their optimal. So if pruning occurs whenever possible then the number of nodes searched will be $b*1*b*1...b*(1||b)$ depending on if the depth is even/odd.

This is because in such a scenario the second players moves do not matter as the respective branches/leafs will be pruned as the first players play will always result in a game-state where alpha is greater than beta hence pruning the search path via the second players move.So we can then say that the best case time complexity is $O(w^{((d/2)} + C))$ where C is a constant whose value depends on if the depth is even or odd.

## 5.3 Complexity of other functions in our code and total complexity

In the image below we list all the vital functions and their complexity. We think the the total complexity is at any given reasonable situation is exponential.[THA17]

As advised by the TA, to calculate the total complexity let's ignore the depth for the moment and do a summation of all the function's complexity.

Total= $O(1)+O(1)+O(w)+O(w^2)+O(w^2)+O(w)+O(w)+O(1)+O(1)+O(1)+O(1)+O(1)+O(1)$
Total Complexity= $8(O(1)) + 2(O(w^2)) + 3(O(w))$
Total Complexity= $2(O(w^2))$

Hence it is still in exponential time and we can now include depth and arrive at the final total worst case complexity for MiniMax with and without alpha beta pruning as : $O(w^d)$
The next section shows a more graphical representation in various cases and situations.

## 5.4 Complexity comparison

We represent the run time complexity in best and worst case situations across various depths. The graph we created below compares MiniMax with and without alpha beta pruning and shows how run time is affected as the board width increases.From the graph we can induce that as the depth increases the impact of alpha-beta pruning is greater on MiniMax in terms of run time complexity.

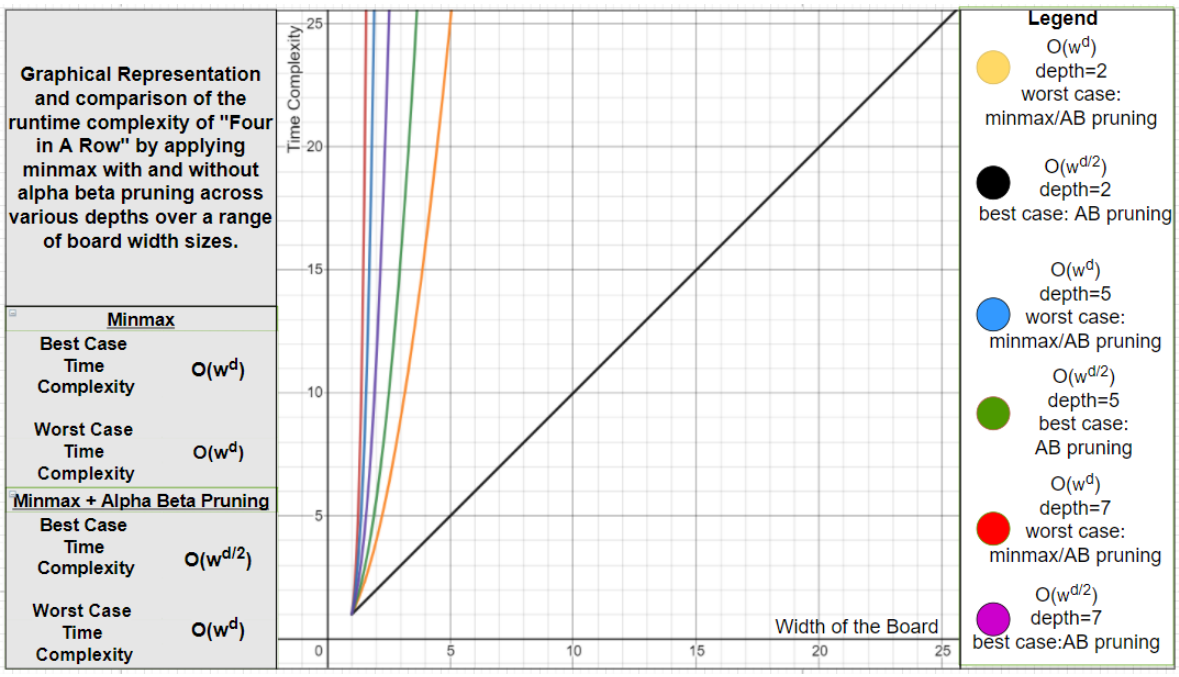



Figure 2: Graphical run time complexity comparison

| Complexity of all major functions | | |
|---|---|---|
| **Class** | **Function** | **Compelxity** |
| Game | startGame | O(1) |
| | isOver | O(1) |
| | winning | O(w) |
| Board | play | O(w) |
| | getNewBoard | O(w^2) |
| SimpleHeristic | evaluate | O(w^2) |
| MinMaxPlayer | miniMax | O(w^d |
| AlphaBetaPlayer | alphaBeta | O(w^d) |
| HumanPlayer | makeMove | O(1) |
| MinMaxNode | getScore | O(1) |
| | getCollumn | O(1) |
| Node | setParent | O(1) |
| | addChild | O(1) |
| | removeChild | O(1) |

Figure 3: Complexity of all major functions

# 6 Interpretation of the Results

We came to the conclusion that, logically, alpha-beta pruning performs much better on larger boards. Furthermore, we found that alpha-beta pruning requires much less evaluations, and therefore performs better, on larger search depths which was also according to our expectations. From the tables we could conclude that N does not have a significant effect on the performance of the algorithm, where as the board size does have a significant effect on the algorithm. This can be explained when looking at the complexity, where the width of the board has impact on the complexity of the algorithm whereas N does not, a larger N could result in more evaluations due to the fact that this increases the difficulty of the game as there are less possible solutions so the game is played for a longer time and thus resulting in more evaluations. When looking at the complexity, we can see that the search depth d has a major impact. This can be supported by the results displayed in the table, where increases in search depth exponentially increases the number of evaluations.

# 7 Conclusion

In conclusion our hypothesis stands true. Using alpha-beta pruning decreased run time and better the complexity of our code exponentially. Our expectations were met as we predicted a trend of higher impact of alpha-beta pruning as the depth increases and the results support our claim. The plots on our graph also do support the trend in complexity as board width and depth is increased which we hypothesized. In the future we may explore options of the impact multidimensional implementation of alpha-beta pruning on multidimensional trees.

# 8 Bonus

We noticed that the Simple Heuristic class provided some evaluations which made the algorithm prefer a move that was non-beneficial for it. Our thought was to come up with an improved version of a Heuristic which also takes into account if there is enough space to finish a row based on the game N and rows (or columns) available, so it doesn't start building up a move in a column or row where there won't be enough columns or rows available to win the game. Then a column scores more points the more available adjacent spaces it has (rows or columns depending on the move).

# References


[DDP22] Mayank Dabas, Nishthavan Dahiya, and Pratish Pushparaj. Solving connect 4 using artificial intelligence. In *International Conference on Innovative Computing and Communications*, pages 727–735. Springer, 2022.

[FK88] Chris Ferguson and Richard E Korf. Distributed tree search and its application to alpha-beta pruning. In *AAAI*, volume 88, pages 128–132, 1988.

[Fri14] Mikael Fridenfalk. N-person minimax and alpha-beta pruning. In *NICOGRAPH International 2014, Visby, Sweden, May 2014*, pages 43–52, 2014.

[THA17] Lukas Tommy, Mardi Hardjianto, and Nazori Agani. The analysis of alpha beta pruning and mtd (f) algorithm to determine the best algorithm to be implemented at connect four prototype. In *IOP Conference Series: Materials Science and Engineering*, volume 190, page 012044. IOP Publishing, 2017.