MinMax
Game Theory

1st Ikic
Mobile Computing
FH OÖ Campus Hagenberg
Hagenberg, Austria
S2010237031@fhooe.at

2nd Shehata
Mobile Computing
FH OÖ Campus Hagenberg
Hagenberg, Austria
S2010237022@fhooe.at

3rd Milosavljevic
Mobile Computing
FH OÖ Campus Hagenberg
Hagenberg, Austria
S2010237014@fhooe.at

***Abstract*—Game theory belongs to the branch of applied mathematics, but is also used in philosophy, life science and social science. Game theory tries to mathematically explain the results of one person, depending on the result or action of another person. One of the most important results was formulated in 1928 by the mathematician John von Neumann and is called the Minimax Theorem. This theorem forms the basis for all subsequent findings in this field.**

I. Introduction

The MiniMax theorem is about finding the optimal game strategy for finite two-person zero-sum games with perfect infromation. With small modifications, it can be used in games such as Go, Chess, Checkers, Four-in-a-row and Tic-Tac-Toe. The MiniMax theorem forms the core of chess programs and chess computers. In 1996, the reigning world chess champion Garry Kasparov was only beaten by brute force calculations. In summary, the basic idea behind the Minimax algorithm is: "If I make this move, my opponent can only make these moves". [1]

II. MiniMax Algorithm

A. Applying MiniMax Algorithm to Connect 4

Here we will explain how the Minimax algorithm works, using the game Connect 4, as you can see in Figure 1, as an example.

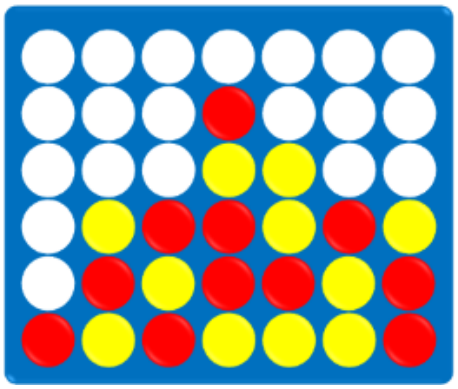

Fig. 1. Connect 4

B. Connect 4 rules of the game

Connect-Four may be a game for two persons. Both players have 21 indistinguishable pieces. Within the standard frame of the game, one set of pieces is yellow and the other set is red. The game is played on a vertical, rectangular board comprising of 7 vertical columns of 6 squares each. If a piece is put in one of the columns, it'll drop down to the last empty square within the column. As soon as a column contains 6 pieces, no other piece can be put within the column. Putting a piece in one of the columns is called: a move. The players make their moves in turn. [2]

Try to get four pieces of your colour into a horizontal, vertical or diagonal row before your opponent! It is not only important to pay attention to your own success, but also to ensure that your opponent does not succeed in completing the row. If all 42 pieces are played and no player has achieved this goal, the game will end in a draw. [3]

Figure 2 shows positions in which White has won the game:

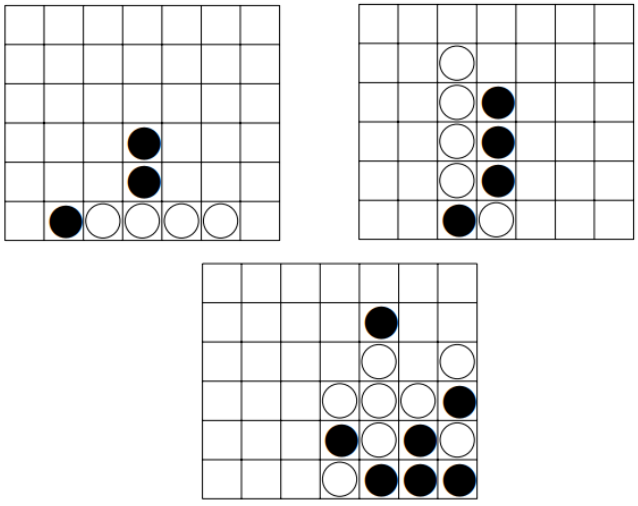

Fig. 2. White has won

A possible draw position is shown in Figure 3

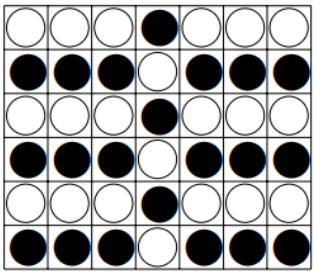

Fig. 3. White has won

C. Mini-Max algorithm and Connect 4

The MinMax algorithm belongs to the backtracking algorithms and can be used in several ways. Basically, it is used in decision and game theory, where it will try to find the optimal move. It is always assumed that our opponent will also choose the optimal move against us. In the MinMax algorithm there are two players. One is called the maximiser and the other one is called the minimiser. The minimiser will try to achieve the lowest score and the maximiser will try to do exactly the opposite. With each move, the state of the board changes. Each board state is therefore assigned a certain value. If the maximiser has the advantage in a certain state in the game, the score of the board will tend to be positive. Conversely, if the minimiser has the advantage in the game at the same state, the value will tend to be negative. [4]

Minimax tree notion	Minimax game notion
Minimax tree	All possible board configurations
Node in the tree	Board configuration
Edge from a max node to a min node	Personal move (move by player max)
Edge from a min node to a max node	Adversary move (move by player min)
Node value	Quality of a given board position
Leaf node	Outcome of a game (either win, tie or lose)
Solution path	Sequence of moves leading to the best outcome

TABLE I
RELATION BETWEEN MINIMAX TREES AND MINIMAX GAMES.

For our MinMax algorithm, a simple game tree would look like the one shown in figure 4. [4]

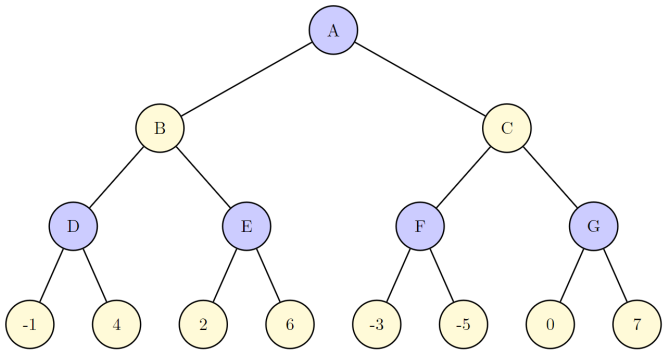

Fig. 4. Simple starting game tree

1) MinMax explanation: Our MinMax algorithm searches the entire depth of the game tree down to the leaf nodes. This search is also called a depth-first search (DFS). The next example will illustrate the algorithm. [5]

First, the algorithm generates the entire game tree and creates the utility values for the end states by applying the utility function as shown in figure 2. For example, in the figure 2 tree diagram, let us assume A as the initial state of the tree. Suppose the maximiser takes the first move, which in the worst case has an initial value equal to negative infinity. Then the minimiser takes the next move, which in the worst case has an initial value equal to positive infinity.

Next, we consider the maximiser with the initial value minus infinity. Each end node is compared with the maximiser's value and finally the maximum value is stored in each maximiser node. For example, let's take the third row (maximiser) from above.

- for node D $\longrightarrow \boldsymbol{max(max(-1,-\infty),4) = 4}$
- for node E $\longrightarrow \boldsymbol{max(max(2,-\infty),6) = 6}$
- for node F $\longrightarrow \boldsymbol{max(max(-3,-\infty),-5) = -3}$
- for node G $\longrightarrow \boldsymbol{max(max(0,-\infty),7) = 7}$

Our tree now looks as shown in the following figure 5.

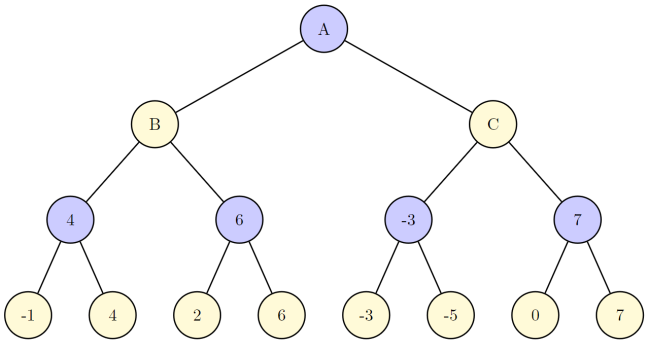

Fig. 5. MinMax game tree

We then compare the values of each node with the value of the minimiser, which is plus infinity.

- for node B $\longrightarrow \boldsymbol{min(min(4,+\infty),6) = 4}$
- for node C $\longrightarrow \boldsymbol{min(min(-3,+\infty),7) = -3}$

Our tree now looks as shown in the following figure 6.

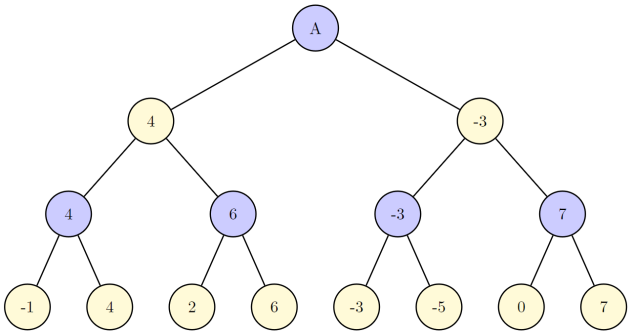

Fig. 6. MinMax game tree

Finally, the maximiser then chooses the maximum value between node B and node C again:

- for node A $\longrightarrow$ $\boldsymbol{max(4,-3)=4}$

This results in the following optimal gameplay also shown in figure 7: A$\rightarrow$ B$\rightarrow$ D$\rightarrow$ I

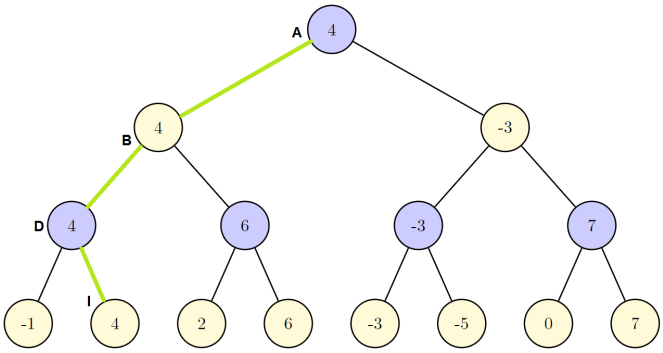

Fig. 7. MinMax game tree

Listing 1. MinMax Pseudocode

```
def minMax(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, minMax(child,
          depth - 1, FALSE))
    return value
  else (* minimizing player *)
    value = +∞
    for each child of node do
       value := min(value, minMax(child,
          depth - 1, TRUE))
  return value
```

Listing 2. MinMax Connect-4 source-code

```
function minimax(state: State, depth = Infinity)
    : Move | undefined {
  const player = state.player;

  const checkIsTerminal = (state: State,
      currentDepth: number): boolean =>
    currentDepth == 0 || state.utility !== 0 ||
        state.moves.length === 0;

  const evalFn = (state: State): number =>
    player === Cell.PLAYER ? state.utility : -
        state.utility;

  function maximizer(state: State, depth: number
      ): number {
    return getActions(state).reduce((prev, action
        ) => {
      const [row, col] = action;
      const best = Math.max(prev, minimizer(
          result(state, action), depth - 1));
      state.board[row][col] = Cell.EMPTY;
      return best;
    }, -Infinity);
  }

  function minimizer(state: State, depth: number
      ): number {
    return getActions(state).reduce((prev, action
        ): number => {
      const [row, col] = action;
      const best = Math.min(prev, maximizer(
          result(state, action), depth - 1));
      state.board[row][col] = Cell.EMPTY;
      return best;
    }, Infinity);
  }

  let best = -Infinity;
  let bestMove: Move | undefined = undefined;

  for (const move of getActions(state)) {
    const [row, col] = move;
    const value = minimizer(result(state, move),
        depth - 1);
    state.board[row][col] = Cell.EMPTY;

    if (value > best) {
      best = value;
      bestMove = move;
    }
  }

  return bestMove;
}
```

III. Optimization technique for MinMax

A. Alpha-Beta pruning

One of the most practiced optimization techniques for the MinMax algorithm is alpha beta pruning. As an optimization technique it only adapts the MinMax algorithm's mechanism and can therefore not be regarded individually. [6] That is to say, the main purpose of this technique is speed enhancement whilst ensuring that no information loss occurs. The MinMax algorithm functions as a Depth first algorithm, starting on the left an traversing through the tree by always visiting the terminating leaf, where the pruning process takes place. A node is being pruned from a tree or sub-tree if it is evident

that the algorithm is not going to take that path. This optimizes runtime from $O(b^d)$to $O(b^d(d/2))$.

In order to further understand this technique it is necessary to look at the basic operations, which are depicted in the following example. The given MinMax tree uses blue nodes to symbolize the maximizing player and yellow for the minimizing player. Additionally the currently regarded node is colored orange and the already pruned nodes grey.

The tree traversal functions as previously explained in section 1. until the orange colored node is reached. There the tree evaluates if it is even necessary to further examine the underlying nodes.

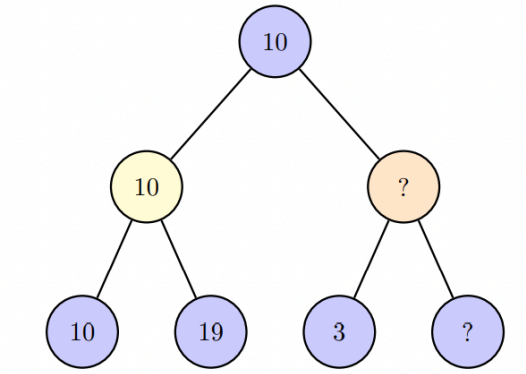

Fig. 8. Alpha-beta pruning

Since the minimizing player always chooses the lowest value it is evident that the orange marked node can only obtain a value ≤ 3. Consequently it is clear that the root node will choose the highest value and therefore will not consider the right branch at all, since it will not provide a value higher than 10.

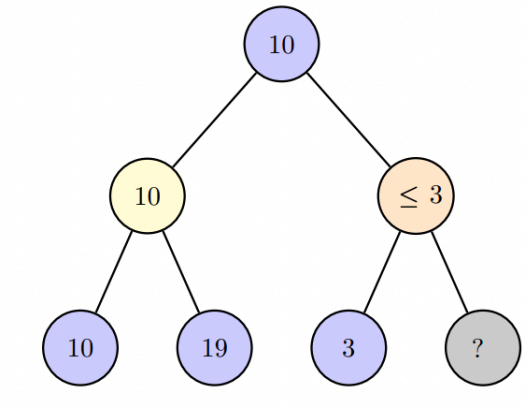

Fig. 9. Alpha-beta pruning

Although pruning evidently shortens the amount of time needed to traverse the tree, it is nonetheless not guaranteed to occur. This becomes clear when switching the leaf node 3 with for instance the value 30. In that case no sub-tree or nodes could have been pruned. This leads to the conclusion that pruning depends on the moves order. Ideally the moves are ordered from best to worst.

With this in mind, let's look further into the algorithm. Before the algorithm is being executed two variables, alpha and beta, have to be added. Alpha is set to negative infinity and beta to infinity, which are the worst possible outcomes for both players.

By entering the tree we move forward until we reach the first node at depth -1. This node owns two child nodes one being 10 and the other 19. Since this is a minimizing node the value 10 is picked. However is not only the node's own value evaluated but also the resulting beta value.

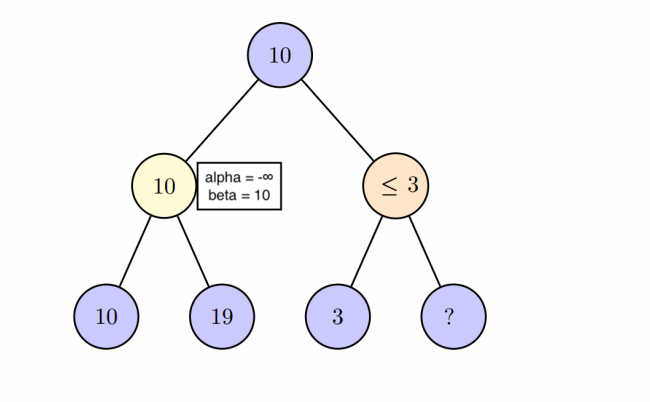

Fig. 10. Alpha-beta pruning

Since beta represents the best value for the minimizing-player at a particular position at the tree and the current node is a minimizing node, beta has to be updated to 10.

The newly calculated values are then passed on to the parent node and switched, resulting in alpha being 10 and beta ∞. These values are then further passed on to the next child node which in our case is at depth -1. To that end, the alpha beta evaluation has to be repeated equivalently to the prior alpha/beta determination. Leading to alpha maintaining the value 10 and beta being updated to 3.

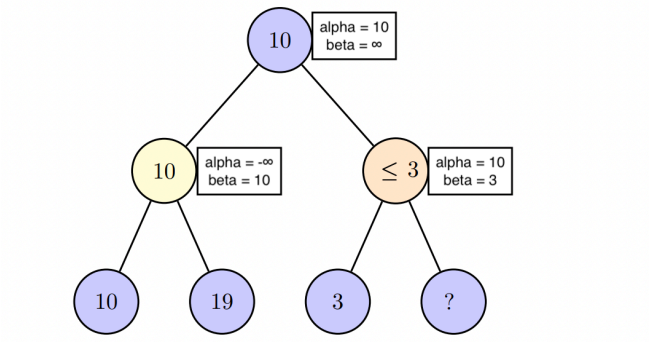

Fig. 11. Alpha-beta pruning

Now the alpha and beta values have to be compared to each other and since beta is $\leq$ alpha, the node can be pruned.

Alpha-Beta Variations

Alpha-beta pruning can be divided into to variations, fail-soft and fail-hard. The difference being that the fail-soft variation updates the alpha and beta value before considering any pruning, whereas the fail-hard prunes before updating any value. The following subsection shows both algorithms.

Listing 3. Fail-hard variation

```
def alphaBeta(node, depth, α, β,
    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, α, β, FALSE))

       if value ≥ β then
          break
       α := max(α, value)
    return value
  else (* minimizing player *)
    value = +∞
    for each child of node do
       value := min(value, alphaBeta(child,
            depth - 1, α, β, TRUE))

       if value ≤ α then
          break
       β := min(β, value)
  return value
```

Listing 4. Fail-soft variation

```
def alphaBeta(node, depth, α, β,
    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, α, β, FALSE))

       α := max(α, value)
       if value ≥ β then
          break
    return value
  else (* minimizing player *)
    value = +∞
    for each child of node do
       value := min(value, alphaBeta(child,
            depth - 1, α, β, TRUE))

       β := min(β, value)
       if value ≤ α then
          break
  return value
```

Listing 5. Alpha-Beta Connect-4 source-code

```
1  function alphaBeta(state: State, depth =
       Infinity): Move | undefined {
2    const player = state.player;
3
4      function checkIsTerminal(state: State,
           currentDepth: number): boolean {
5      return currentDepth > depth || state.utility
           !== 0 || state.moves.length === 0;
6    }
7
8    function evalFn(state: State): number {
9      return player === Cell.PLAYER ? state.utility
           : -state.utility;
10   }
11
12   function maximizer(state: State, depth: number
         , alpha: number, beta: number): number {
13     if (checkIsTerminal(state, depth)) return
           evalFn(state);
14
15     let best = -Infinity;
16     const { board } = state;
17
18     for (const move of getActions(state)) {
19       const [row, col] = move;
20
21       best = Math.max(best, minimizer(result(
             state, move), depth + 1, alpha, beta));
22       board[row][col] = Cell.EMPTY;
23
24       if (best >= beta) return best;
25       alpha = Math.max(alpha, best);
26     }
27
28     return best;
29   }
30
31   function minimizer(state: State, depth: number
         , alpha: number, beta: number): number {
32     if (checkIsTerminal(state, depth)) return
           evalFn(state);
33
34     let best = Infinity;
35     const { board } = state;
36
37     for (const move of getActions(state)) {
38       const [row, col] = move;
39
40       best = Math.min(best, maximizer(result(
             state, move), depth + 1, alpha, beta));
41       board[row][col] = Cell.EMPTY;
42
43       if (best <= alpha) return best;
44       beta = Math.min(beta, best);
45     }
46
47     return best;
48   }
49
50     const { board } = state;
51
52   let best = -Infinity;
```

```
53    let bestMove: Move | undefined = undefined;
54
55    for (const move of getActions(state)) {
56      const [row, col] = move;
57
58      const value = minimizer(result(state, move),
            1, best, Infinity);
59      board[row][col] = Cell.EMPTY;
60
61      if (value > best) {
62        best = value;
63        bestMove = move;
64      }
65    }
66
67    return bestMove;
68  }
```

B. Further optimization

Additionally to the alpha-beta pruning there are other options to improve the algorithm's run time, which can be summarized to ordering heuristics, aspiration search and Killer heuristics. [7] [6]

1) Ordering heuristic: Ordering heuristic can be used to figure out whether a move is going to potentially cause an alpha beta cutoff.Inner nodes, which are essentially all nodes except the root node or leaf nodes, are ordered accordingly to optimize the algorithm's run time. The best ordering for the inner nodes is logic based and can be calculated. Some techniques include iterative deepening, transposition tables and refutation tables. [7]

2) Aspiration search : This technique narrows down the window in which the search is taking place. To put it another way, the search window is determined by the alpha and beta difference. Meaning that a low difference between these values can lead to a smaller search window which would subsequently result in better search performances. However if the search does not lead to the desired outcome the search window has to be broaden and the search restarted. [7]

3) Killer heuristic : Killer heuristics are based on making use of already existing information. To that end, a node's subtree information is appraised and stored for potential use. If a certain node causes a cutoff, this move will be examined first when traversing the tree again. The information is collectively regarded for each depth level. For example, if a node at depth 2 triggers the most cutoffs it is most likely going to cause similar effects the second time the tree is searched. Researches have proven that even though reordering inner nodes takes up time resources, it still leads to a overall better performance. In general, knowledge based on the application is used to determine more use case specific heuristics. If the information's quality is not sufficient enough implementation will most likely add calculated search evaluations to the evaluation. Nevertheless would even an imperfect heuristic lead to better results. [7]

IV. Related work

MinMax algorithm is undoubtedly not the first algorithm created for simultaneous move games. Over the past decades there have been multiple adaptations to the problem, which can be summarize into the following three categories. [8]

A. Iterative learning algorithms

Iterative learning algorithms are based on self-play strategies, which are determined by iterating through the tree.

In 2000 new adaptations for this algorithm were presented,including the no-regret algorithm, which was then applied to One-Card Poker. Additionally a counterfactual regret minimization (CFR) was introduced, which generally speaking finds it's use in games obtaining a larger amount of information. Subsequently CFR was used for Poker AI and resulted in an algorithm efficient enough for solving Head up limit Texan Hold'em. [9]

A variation of this technique called iterative deepening can be used for MinMax optimization. Hereby the search tree is traversed ply after ply, which is a half move. After each move the depth value gets incremented. Also every iteration has a certain amount of time allocated to it in which it should execute the iteration. If the time runs out and the process has not finished yet, it gets terminated and has to be executed again. The algorithms main goal is to increase the probability of searching the best move from the root onwards. Meaning that the best path is already picked at the beginning. (often used for depth first search algorithms)

B. Exact backward induction algorithms

Exact backward induction algorithms are based on a recursive iteration of the tree, which obtains information based on a node's child value. This technique is mostly used in sequential games but can be altered in order to suit simultaneous move games.

C. Approximative sampling algorithms

Approximate sampling algorithms generate samples according to a given distribution. These examples can be used to further predict the probability of potential moves. [10]
A common algorithm that uses this technique is called Monte Carlo Tree Search(MCTS), which is utilized in extensive-form games. [8] These make use of a certain tree search which also considers the amount of time needed to actually choose a move. The MCTS's functionality was proven by the success of the game Go and was later on adapted with UCB (upper confidence bounds) which was then used in the game Tron. [11]

V. Conclusion

In this paper, the main principles of the MinMax algorithms were analysed. We did not only look at the function itself but also into optimization techniques such as alpha/beta pruning and further techniques which included ordering heuristics, aspiration search and killer heuristics. Additionally we looked into related algorithms and there use cases.

REFERENCES

[1] B. Michael. (2020, 8) Introduction to minimax game theory. [Online]. Available: https://www.cosy.sbg.ac.at/~held/teaching/wiss_arbeiten/slides_19-20/KI_in_Videospielen.pdf

[2] V. Allis. (1988, 10) Connect 4 rules of the game. [Online]. Available: http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf

[3] N. S. Howard Wexler. (2020, 8) Connect 4 rules of the game. [Online]. Available: https://www.spielregeln-spielanleitungen.de/spiel/vier-gewinnt/

[4] E. Hussain. (2020, 1) Minimax game theory for connect 4. [Online]. Available: https://www.academia.edu/41561708/Minimax_with_alpha_beta_pruning_connect_4_game_

[5] K. Jonathan C.T. (2020, 10) Minimax game theory for connect 4. [Online]. Available: https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f

[6] R. Nasa, R. Didwania, S. Maji, and V. Kumar, "Alpha-beta pruning in mini-max algorithm - an optimized approach for a connect-4 game," *International Research Journal of Engineering and Technology*, vol. 05, no. 04, pp. 1637–1641, 2018. [Online]. Available: https://www.science.org/doi/10.1126/science.1259433

[7] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 11, no. 11, pp. 1203–1212, 1989.

[8] B. Boansk´y, M. L. Viliam Lisý, J. erm´ak, and M. H. Winands, "Algorithms for computing strategies in two-player simultaneous move games," *ScienceDirect*, vol. 237, pp. 1–40, 2016. [Online]. Available: https://reader.elsevier.com/reader/sd/pii/S0004370216300285

[9] M. Bowling, N. Burch, M. Johanson, and O. Tammelin, "Heads-up limit hold'em poker is solved," *Science*, vol. 347, no. 6218, pp. 145–149, 2015. [Online]. Available: https://www.science.org/doi/10.1126/science.1259433

[10] C. Swamy and D. B. Shmoys, "Sampling-based approximation algorithms for multi-stage stochastic optimization," in *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*. IEEE, 2005, pp. 357–366. [Online]. Available: https://www.math.uwaterloo.ca/~cswamy/papers/multistagefnl.pdf

[11] S. Samothrakis, D. Robles, and S. M. Lucas, "A uct agent for tron: Initial investigations," *IEEE Symposium on Computational Intelligence and Games, CIG*, pp. 365–371, 2010.