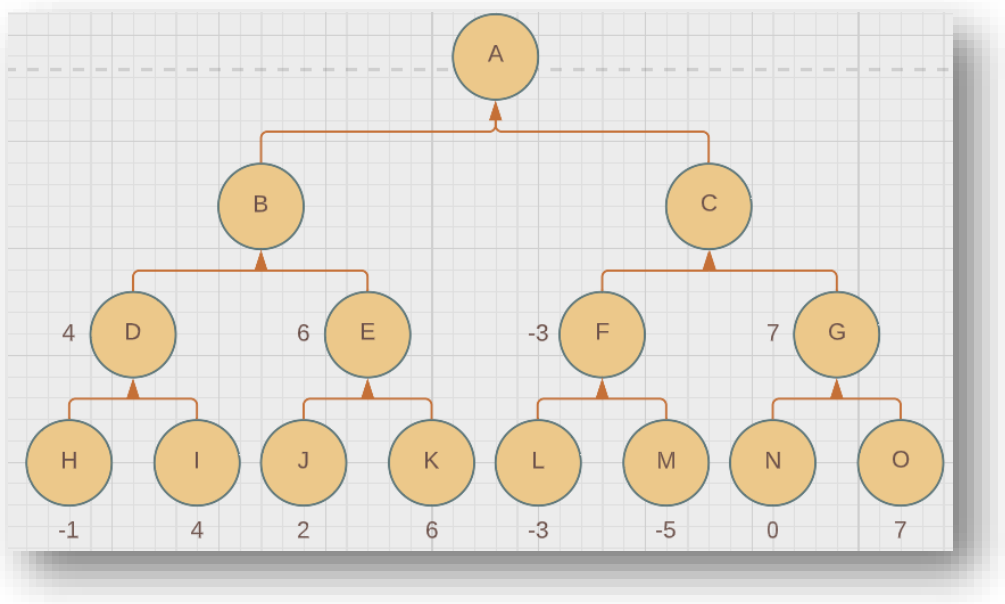
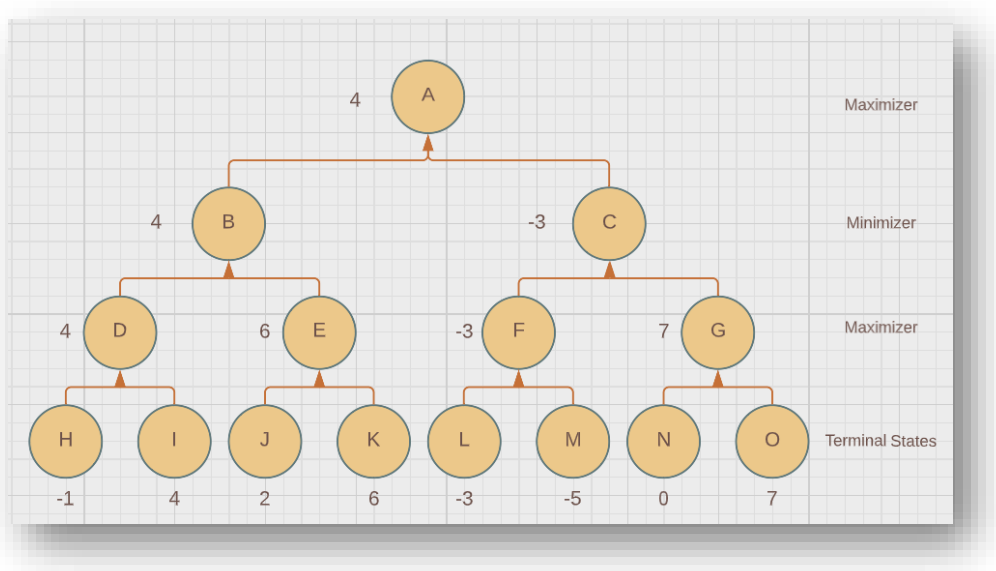
Artificial Intelligence Notes

As your classmate (or a group) presents, take your notes here:

Day:	26 December 2022
Topic:	Min Max Algorithm
Key Message:	Minimizes the possible loss in worst case and finds the best score using comparison between different values.
1st point:	Makes decision using backtracking, e.g., optimal moves in games. Maximizer and Minimizer are the two players of Minimax where Maximizer tries to get highest score possible and Minimizer does the opposite, minimizes the Maximizer's score. • Evaluate Game Trees • Evaluator/Utility Function to generate values at leaf/terminal nodes which are used to determine best score. • 2 Players deterministic play
2nd point:	Minimax follows Depth First Search Technique, so in this game-tree, we must go all the way through the leaves to reach the terminal nodes. At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. • Minimize the possible loss in worst case (Maximum loss) • Maximize the minimum gain

	Minmax algorithm is used in: • AI • Decision Theory • Game Theory • Statistics • Philosophy
3rd point:	Root node to terminal nodes, which contain terminal values so that we can compare them while doing backtracking, it will produce moves for Maximizer and Minimizer.
4th point:	Step 1: Generate game tree and apply utility function. This utility function will generate terminal values for terminal nodes. Worst minimum score for Maximizer can be minus infinity, which is its initial value and worst maximum score for Minimizer can be plus infinity, which is its initial value. Both Min and Max will compare the new values with initial values first. First Move: Maximizer Next Move: Minimizer
5th point:	Step 2: First we will find utility value of Maximizer and then compare the initial values with all the values in terminal state and determine the higher node value.

	 For D: max (-1, minus infinity) => max (-1, 4) = 4 For E: max (2, minus infinity) => max (2, 6) = 6 For F: max (-3, minus infinity) => max (-3, -5) = -3 For G: max (0, minus infinity) => max (0, 7) = 7
6th point:	Step 3: Minimizer turn to compare the values with its initial state.

	 For B: min (4, plus infinity) => min (4, 6) = 4 For C: min (-3, plus infinity) => min (-3, 7) = -3
7th point:	Step 4: After Minimizer turn its again Maximizer turn who will choose the maximum value from the given node. For A: max (4, minus infinity) => max (4, -3) = 4 In this example we just had four layers but in real life there can be uncountable number of layers, but the core concept will remain the same which is comparing values.
8th point:	Limitations: • Not always feasible to traverse whole tree • Time limitation (turns) Key Improvements: • Evaluation instead of Utility Function

	• Evaluation Function provides estimation of utility at given position
Important vocabulary :	• Minimizer and Maximizer • Terminal Values using Utility Functions • Comparison of values for best score
Questions	• What is Minmax Algorithm? • What is Minmax Problem? • Minmax in Artificial Intelligence

What is the Minimax algorithm?

It is used in games such as tic-tac-toe, go, chess, Isola, checkers, and many other two-player games. Such games are called games of perfect information because it is possible to see all the possible moves of a particular game. There can be two-player games which are not of perfect information such as Scrabble because the opponent 's move cannot be predicted. It is like how we think when we play a game: "if I make this move, then my opponent can only make only these moves," and so on. Minimax is called so because it helps in minimizing the loss when the other player chooses the strategy having the maximum loss.

Terminology

- **Game Tree:** It is a structure in the form of a tree consisting of all the possible moves which allow you to move from a state of the game to the next state.

A game can be defined as a search problem with the following components:

- **Initial state:** It comprises the position of the board and showing whose move it is.
- **Successor function:** It defines what the legal moves a player can make are.
- **Terminal state:** It is the position of the board when the game gets over.
- **Utility function:** It is a function which assigns a numeric value for the outcome of a game. For instance, in chess or tic-tac-toe, the outcome is either a win, a loss, or a draw, and these can be represented by the values +1, -1, or 0, respectively. There are games that have a much larger range of possible outcomes; for instance, the utility in backgammon varies from +192 to -192. A utility function can also be called a payoff function.
- **Evaluation function:** Minimax assumes that we can search all the way to the terminal states, this is not often practical. Instead of doing this we could use a heuristic evaluation function to estimate which moves leads to better terminal states. Now a cut-off test must be used to limit the search. Choosing a good evaluation function is very important to the efficiency of this method. The evaluation function must agree with the utility function for terminal states, and it must be a good predictor of the terminal values. If the evaluation function is infallible then no search is necessary, just choose the move that gives the best position. The better the evaluation function the less search need to be done.

Minimax Algorithm Pseudocode

Minimax consists of 5 steps.

1. Generate the complete game tree.
2. Apply the utility function to all the terminal states.
3. Use the utility of the terminal states to calculate a utility value for their parents (either max or min) depending on depth.

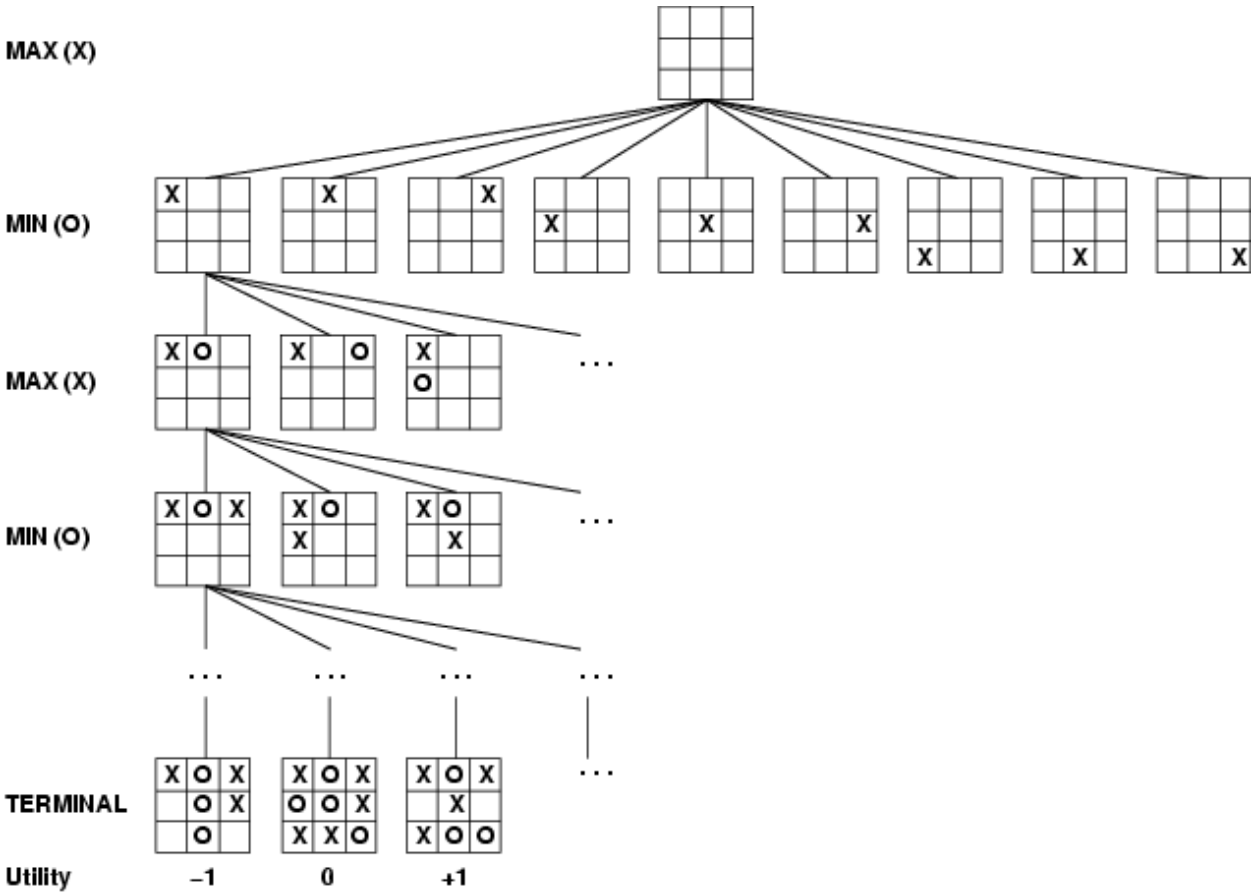
4. Continue up to root node.
5. Choose to move with highest value from root node.

```
function MINIMAX-DECISION(game) returns an operator

    for each op in OPERATORS[game] do
        VALUE[op] ← MINIMAX-VALUE(APPLY(op, game), game)
    end
    return the op with the highest VALUE[op]

--------------------------------------------------------------

function MINIMAX-VALUE(state, game) returns a utility value

    if TERMINAL-TEST[game](state) then
        return UTILITY[game](state)
    else if  MAX is to move in state then
        return the highest MINIMAX-VALUE of SUCCESSORS(state)
    else
        return the lowest MINIMAX-VALUE of SUCCESSORS(state)
```

How does the algorithm work?

There are two players involved in a game, called MIN and MAX. The player MAX tries to get the highest possible score and MIN tries to get the lowest possible score, i.e., MIN and MAX try to act opposite of each other. The general process of the Minimax algorithm is as follows:

Step 1: First, generate the entire game tree starting with the current position of the game all the way up to the terminal states. This is how the game tree looks like for the game tic-tac-toe.



Let us understand the defined terminology in terms of the diagram above.

1. The initial state is the first layer that defines that the board is blank it's MAX's turn to play.
2. Successor function lists all the possible successor moves. It is defined for all the layers in the tree.
3. Terminal State is the last layer of the tree that shows the final state, i.e., whether the player MAX wins, loses, or ties with the opponent.
4. Utilities in this case for the terminal states are 1, 0, and -1 as discussed earlier, and they can be used to determine the utilities of the other nodes as well.

Step 2: Apply the utility function to get the utility values for all the terminal states.

Step 3: Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes.

Step 4: Calculate the utility values with the help of leaves considering one layer at a time until the root of the tree.

Step 5: Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX must choose the highest value.

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