Artificial Intelligence USDS1402

Module 1 Chapter 3

Adversarial Search

ADVERSARIAL SEARCH

Games

Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games.

Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Terms of Games in AI

- Deterministic Games: In deterministic games, there are **no random elements** (like dice rolls or shuffled cards). The outcome of a move is **completely determined** by the current state and the players' actions. Examples: **Chess, Checkers, Go, Othello.** These games have **no chance moves** — the next state depends only on the players' decisions. Both players can see the **entire game state** (positions of all pieces), making them **perfect-information deterministic games**.
- Games with Chance Moves: These games include **random elements** such as dice rolls, shuffled cards, or random draws that affect the outcome. Examples: **Backgammon, Monopoly.** In **Backgammon**, dice rolls determine how far pieces move — introducing randomness. In **Monopoly**, dice decide movement and cards ("Chance", "Community Chest") add further randomness. So, these are **stochastic (non-deterministic)** games —outcomes depend on **both players' actions and chance**.
- Perfect Information Games: Every player knows the **complete game state** at all times. There is **no hidden information** — everyone can see everything relevant. **Examples: Chess, Checkers, Go, Othello.** Players can see all pieces and possible moves. No hidden cards or secret information.
- Imperfect Information Games: Some information about the game state is **hidden** from one or more players. Players must make decisions **without full knowledge** (e.g., hidden cards, unknown moves). **Examples: Battleships, Blind Tic-Tac-Toe, Bridge, Poker, Scrabble, NuclearWar.** In **Battleships**, you can't see the opponent's ship positions. In **Blind Tic-Tac-Toe**, you play without seeing the opponent's marks. In **Bridge**, **Poker**, and **Scrabble**, cards or tiles are hidden. **Nuclear War** (strategy game) involves secret planning and incomplete knowledge. Hence, these are **imperfect-information games**.

Types of Games in AI

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind tic-tac-toe	Bridge, poker, scrabble, nuclear war

Zero-Sum Game

Zero-sum games are adversarial search which involves pure competition.

In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.

One player of the game try to maximize one single value, while other player tries to minimize it.

Each move by one player in the game is called as ply.

Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

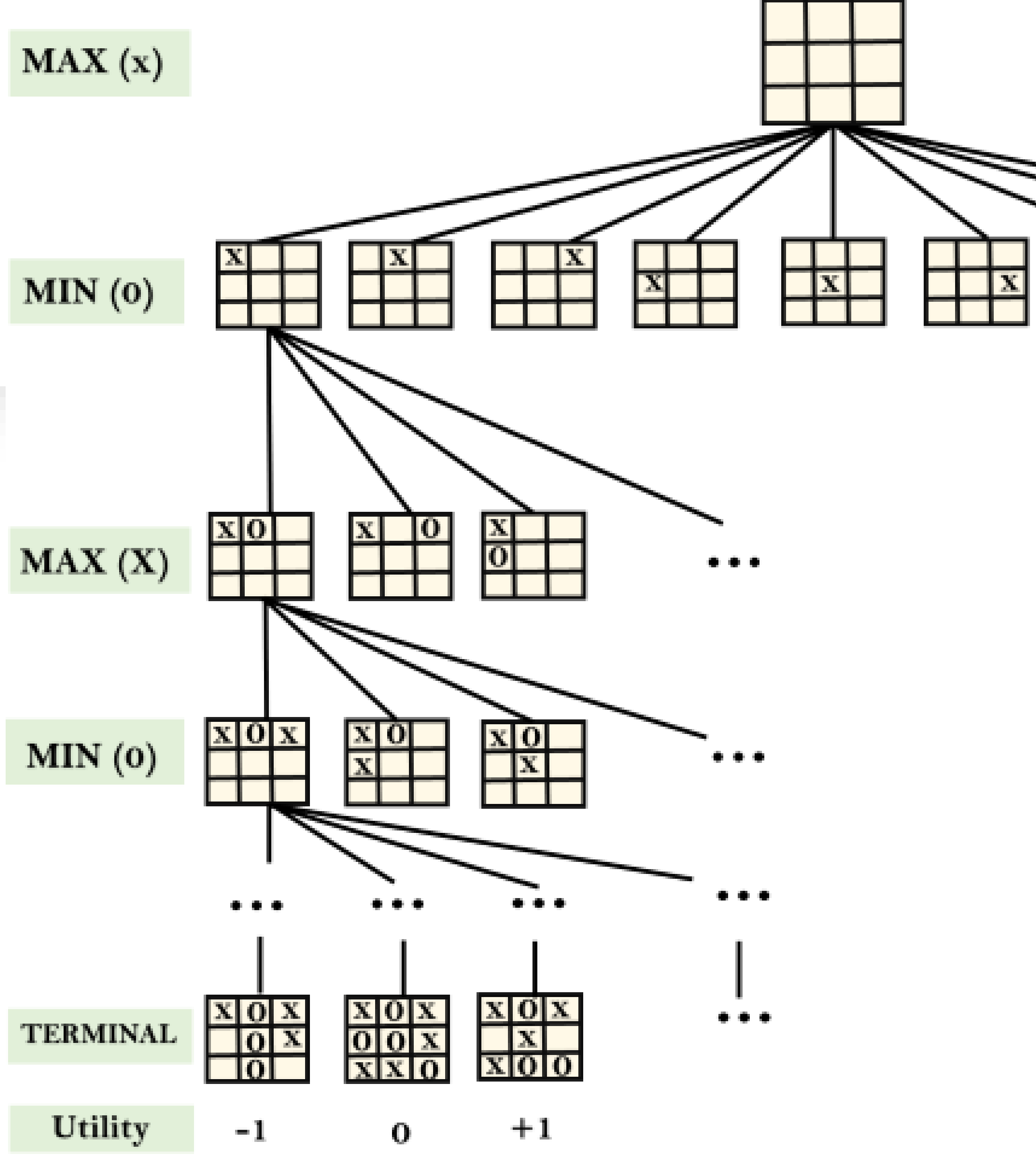
Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

Game tree

- A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, action function, and result Function.
- **Example: Tic-Tac-Toe game tree:**
- The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:
- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



Example Explanation:

From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.

Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.

Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.

So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.

In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Optimal Decisions in Games

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state { a terminal state that is a win.

In adversarial search,

- MIN has something to say about it.
- MAX must nd a strategy, which species MAX's move in the initial state, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN to those moves, and so on.

Optimal Decisions in Games - MinMax

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.
- In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN.} \end{cases}$$

Min-Max Algorithm in Artificial Intelligence

Min-Max Algorithm in Artificial Intelligence

Min-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.

Min-Max algorithm uses recursion to search through the game-tree.

Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various two-players game. This Algorithm computes the minimax decision for the current state.

In this algorithm two players play the game, one is called MAX and other is called MIN.

Min-Max Algorithm in Artificial Intelligence

- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minmax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minmax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of Min-Max Algorithm

In the two-player game example, there are two players one is called Maximiser and other is called Minimizer.

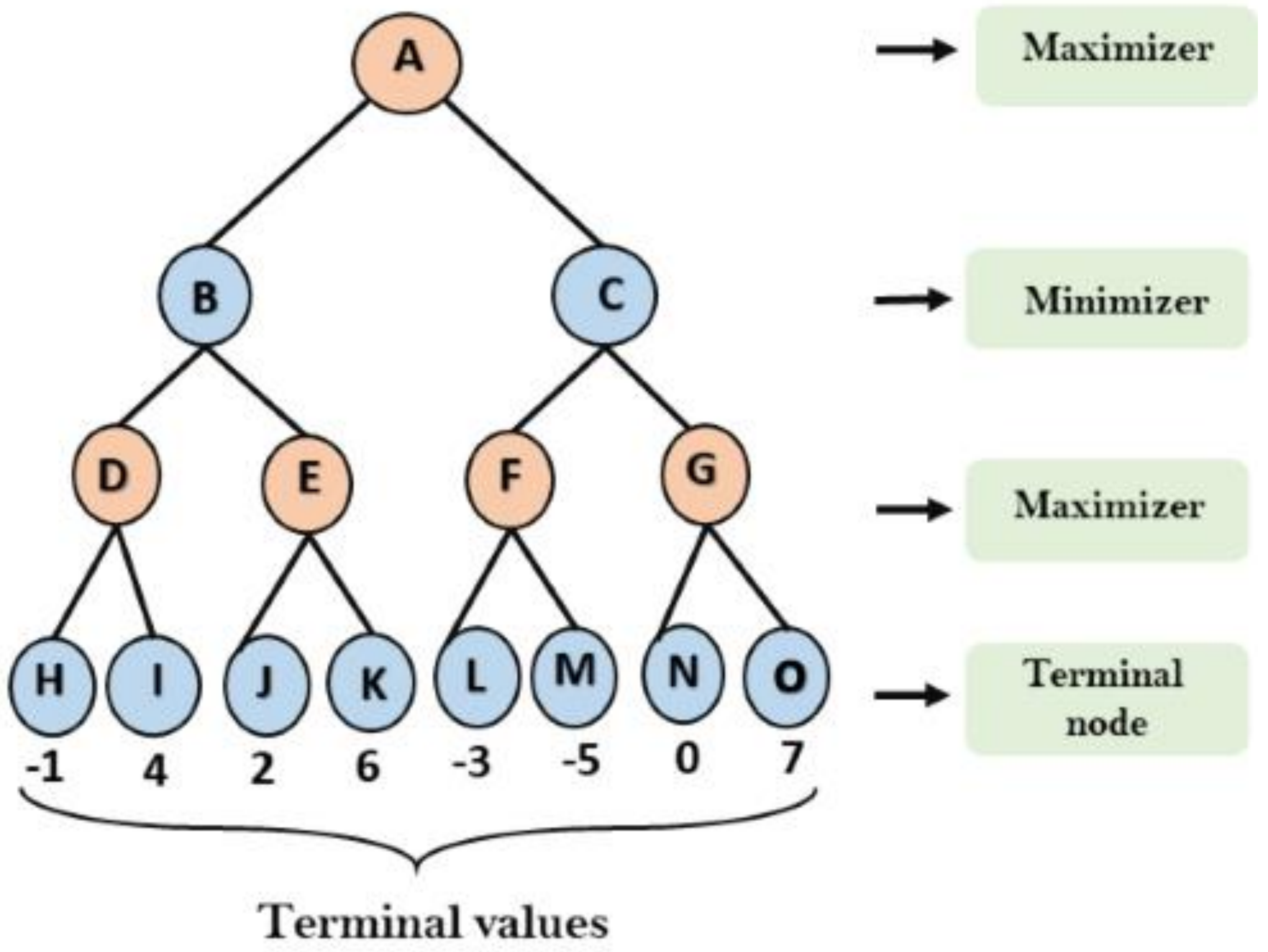
Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.

This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.

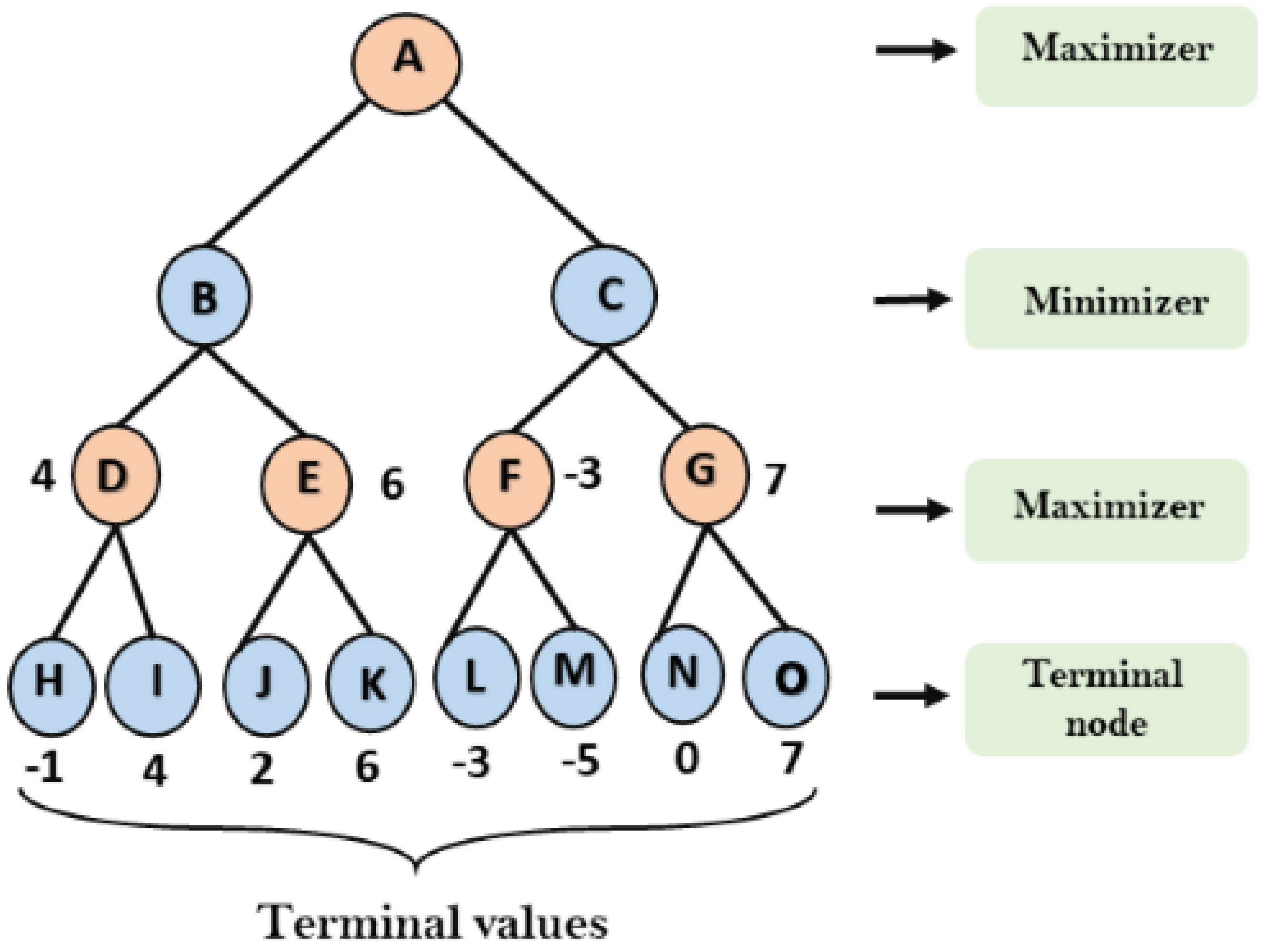
At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs.

Following are the main steps involved in solving the two-player game tree

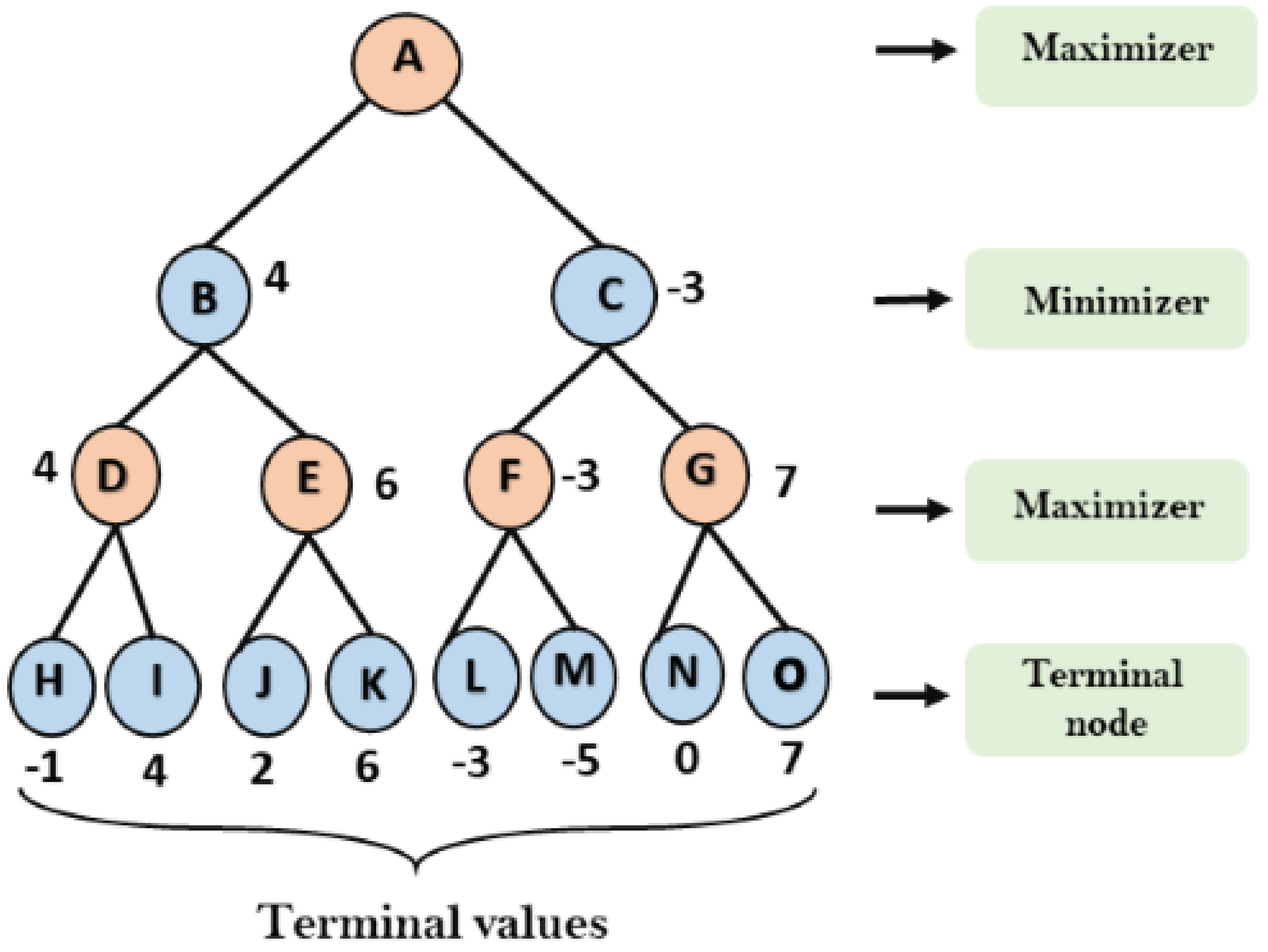
- **Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



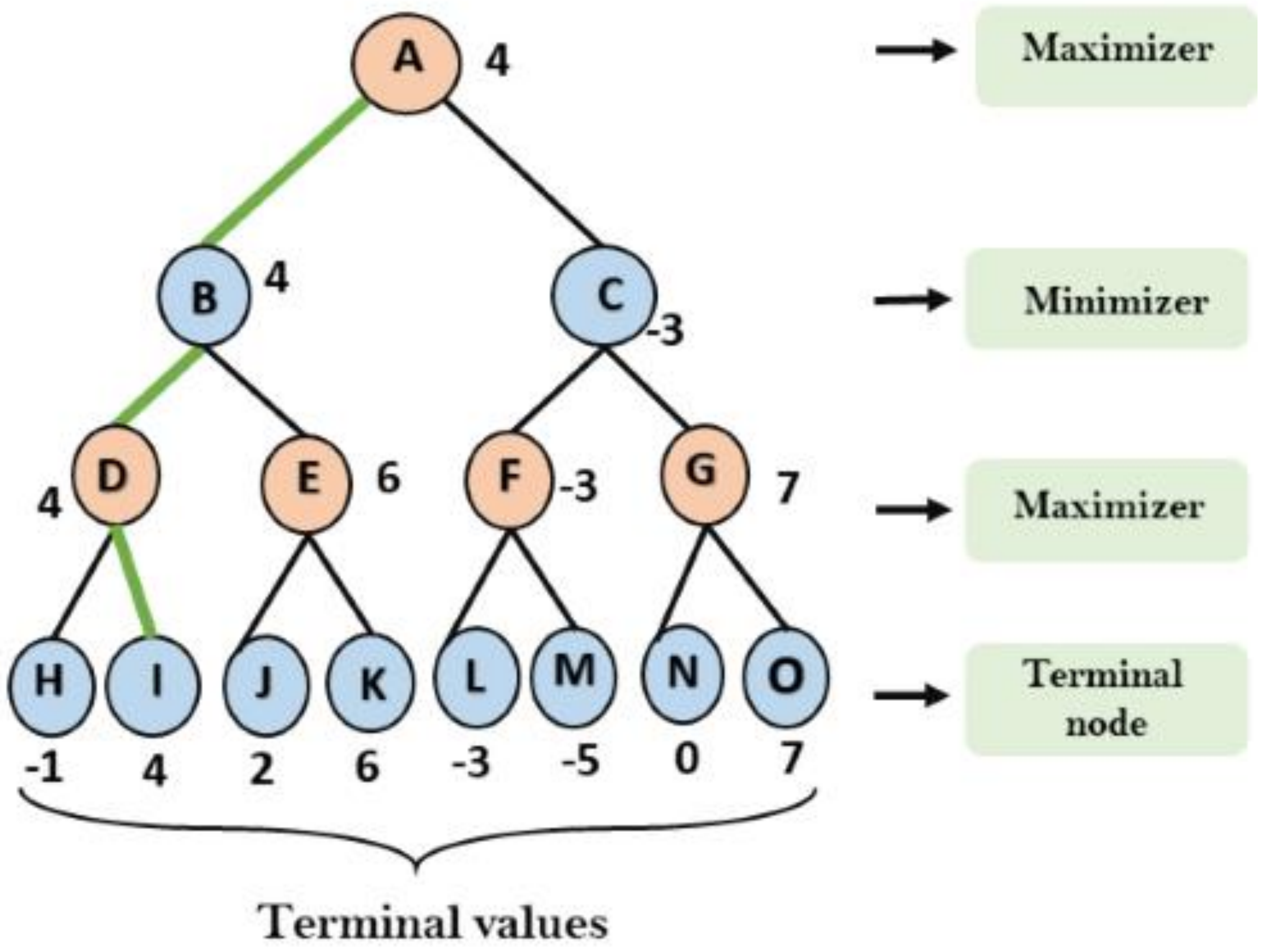
- **Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.
- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



- **Step 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



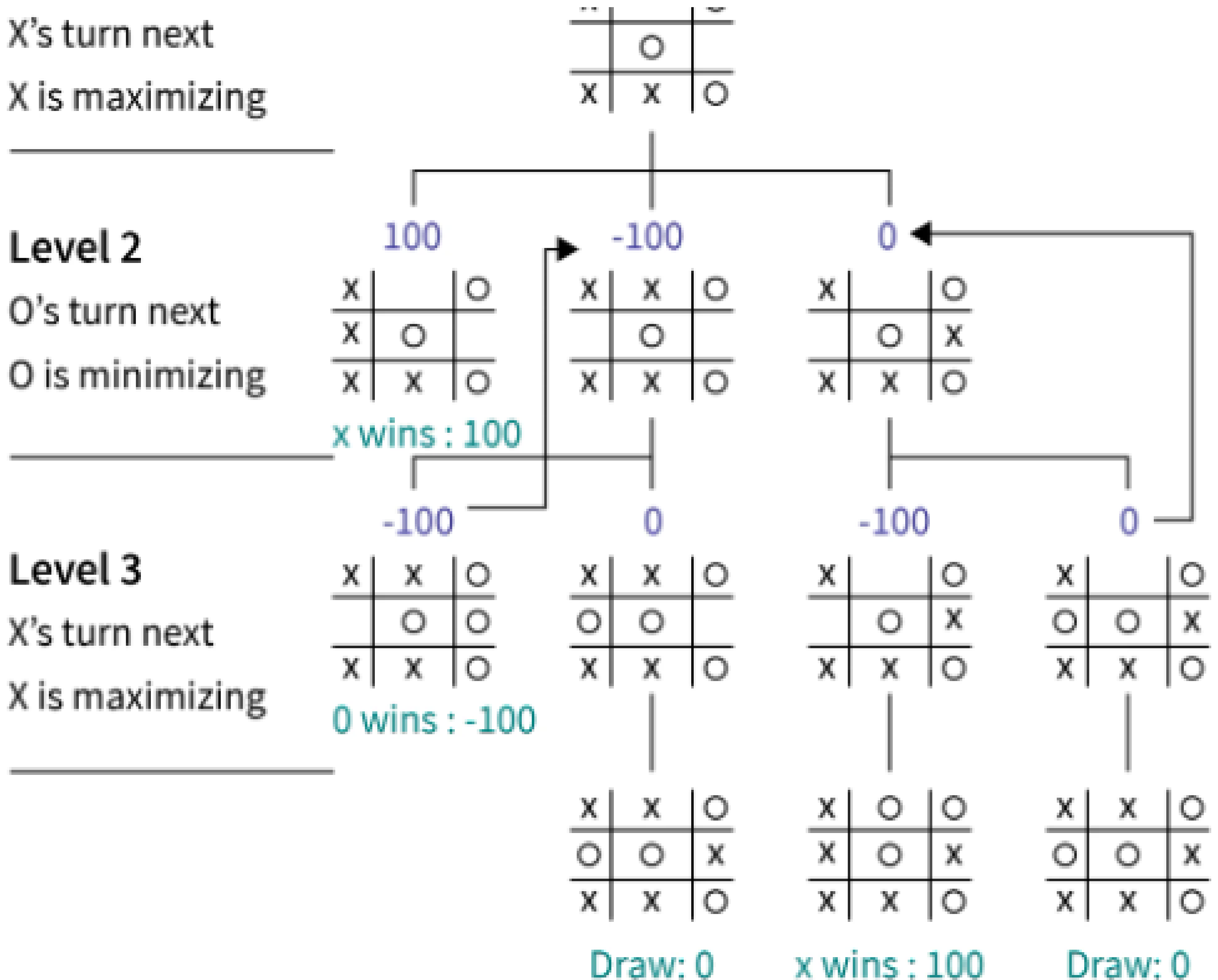
- **Step 4:** Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
- For node A max(4, -3)= 4



Pseudocode/ Algorithm for MinMax

```
function minimax(node, depth, maximizingPlayer):
    if depth = 0 or node is a terminal node:
        return the heuristic value of the node
    if maximizingPlayer:
        bestValue = -infinity
        for each child node of node:
            v = minimax(child, depth - 1, FALSE)
            bestValue = max(bestValue, v)
        return bestValue
    else:
        bestValue = +infinity
        for each child node of node:
            v = minimax(child, depth - 1, TRUE)
```

Example


X's turn next
X is maximizing
Level 2
O's turn next
O is minimizing
Level 3
X's turn next
X is maximizing
100
-100
0
x wins : 100
-100
0
-100
0
0 wins : -100
Draw: 0
x wins : 100
Draw: 0

Limitation of the minimax Algorithm

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc.

This type of games has a huge branching factor, and the player has lots of choices to decide.

This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.