Game Playing

Module 3

Two Player Zero Sum Games

- A **Two-Player Zero-Sum Game** is a mathematical representation of a competitive situation where:
- **Two players** compete against each other.
- **One player's gain is equal to the other player's loss** (sum = 0).
- Used in **game theory**, AI decision-making, and economics.

Two Player Zero Sum Games

- **Two Players** – One is the **Maximizing Player**, and the other is the **Minimizing Player**.
- **Zero-Sum Property** – If one player gains +10, the other loses -10, so the sum is **0**.
- **Perfect Information** – Both players know all possible moves (e.g., Chess, Tic-Tac-Toe)
- **Strategy-Based** – Players make decisions to **maximize** their advantage and **minimize** the opponent's.

- Chess
- Tic-Tac-Toe
- Rock Paper Scissor

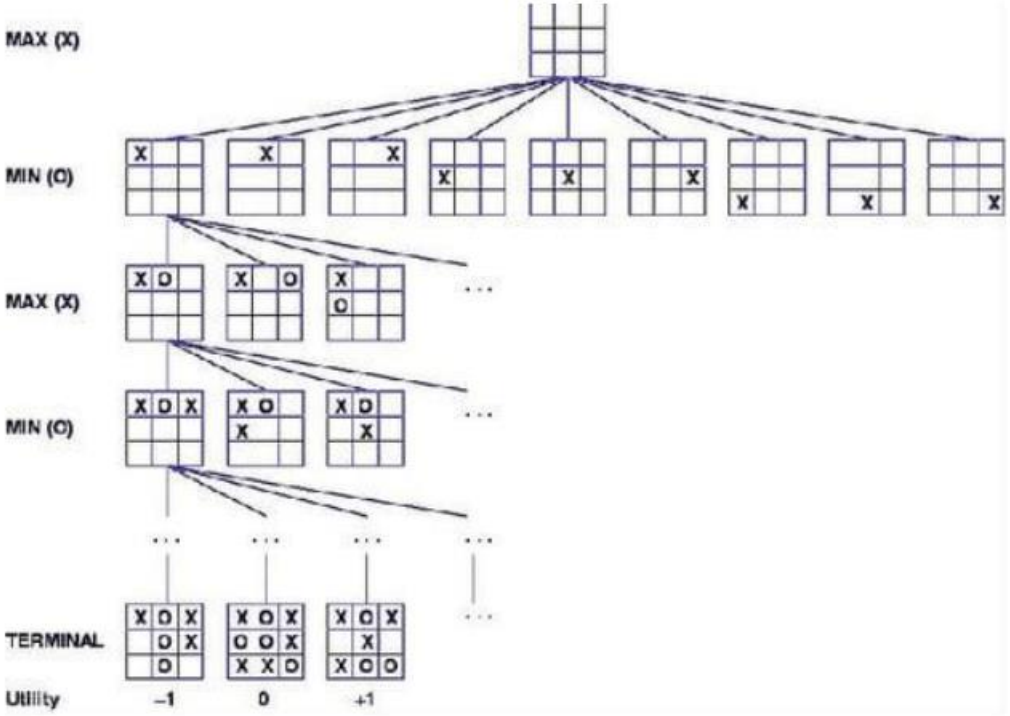
Rock Paper Scissor

	Player 2			
Player 1		R	P	S
	R	(0,0)	(-1, 1)	(1, -1)
	P	(1, -1)	(0,0)	(-1, 1)
	S	(-1, 1)	(1, -1)	(0,0)

Modeling Two Player Zero Sum Games as search problems

- A **search problem** consists of:
- **Initial State** – The starting position of the game.
- **Players (Agents)** – **Maximizing Player (AI)** and **Minimizing Player (Opponent)**.
- **Actions** – The set of legal moves available at each state.
- **State Transition Function** – Defines how the game state changes after a move.
- **Goal State** – A state where the game ends (win, lose, or draw).
- **Utility Function** – Assigns a numerical value to terminal states to determine the winner.

Game Tree-Tic tac toe

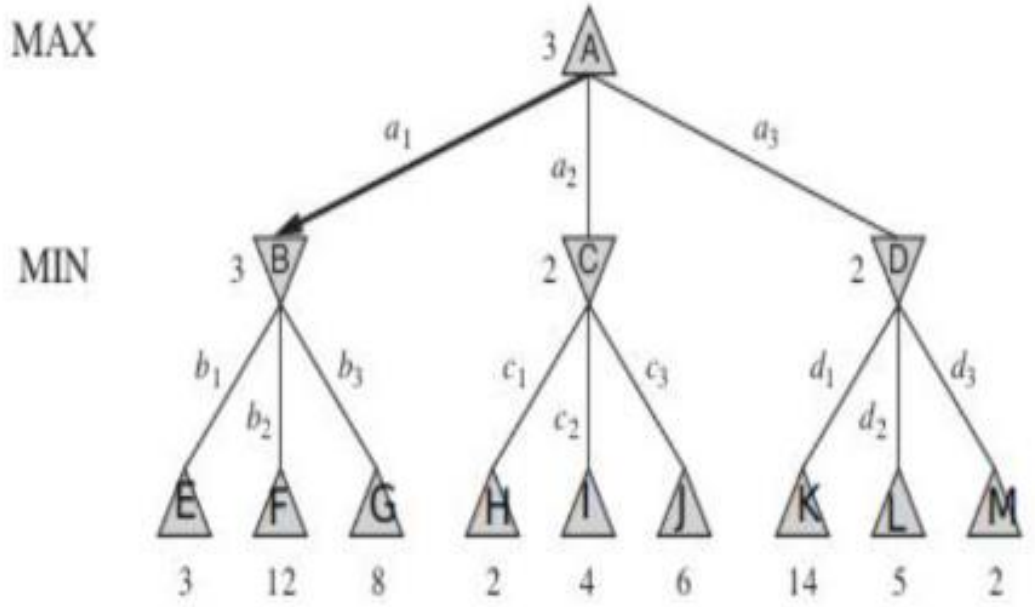


- Since **one player's gain is the other's loss**, the **Minimax algorithm** is perfect for these games.
- **Maximizing Player (AI):** Tries to get the highest score.
- **Minimizing Player (Opponent):** Tries to lower the AI's score.

Minimax Algorithm

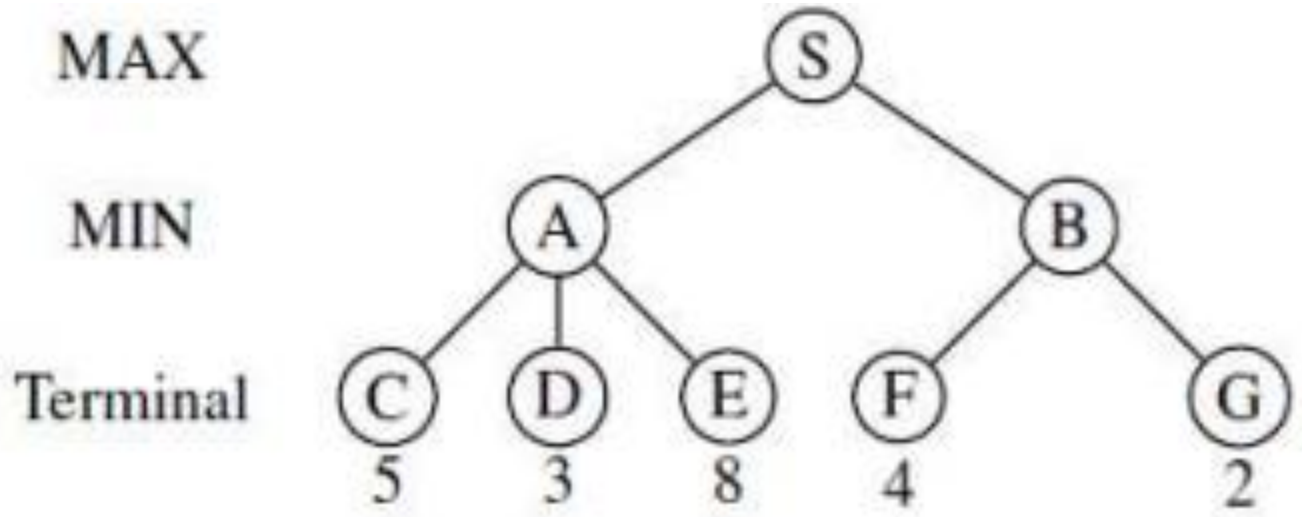
1. Expands the entire game tree up to the terminal states.
2. Evaluates states using a **utility function**.
3. **Backpropagates values**:
   - Maximizing Player picks the **maximum** value.
   - Minimizing Player picks the **minimum** value.

- Solve the following zero sum two player game

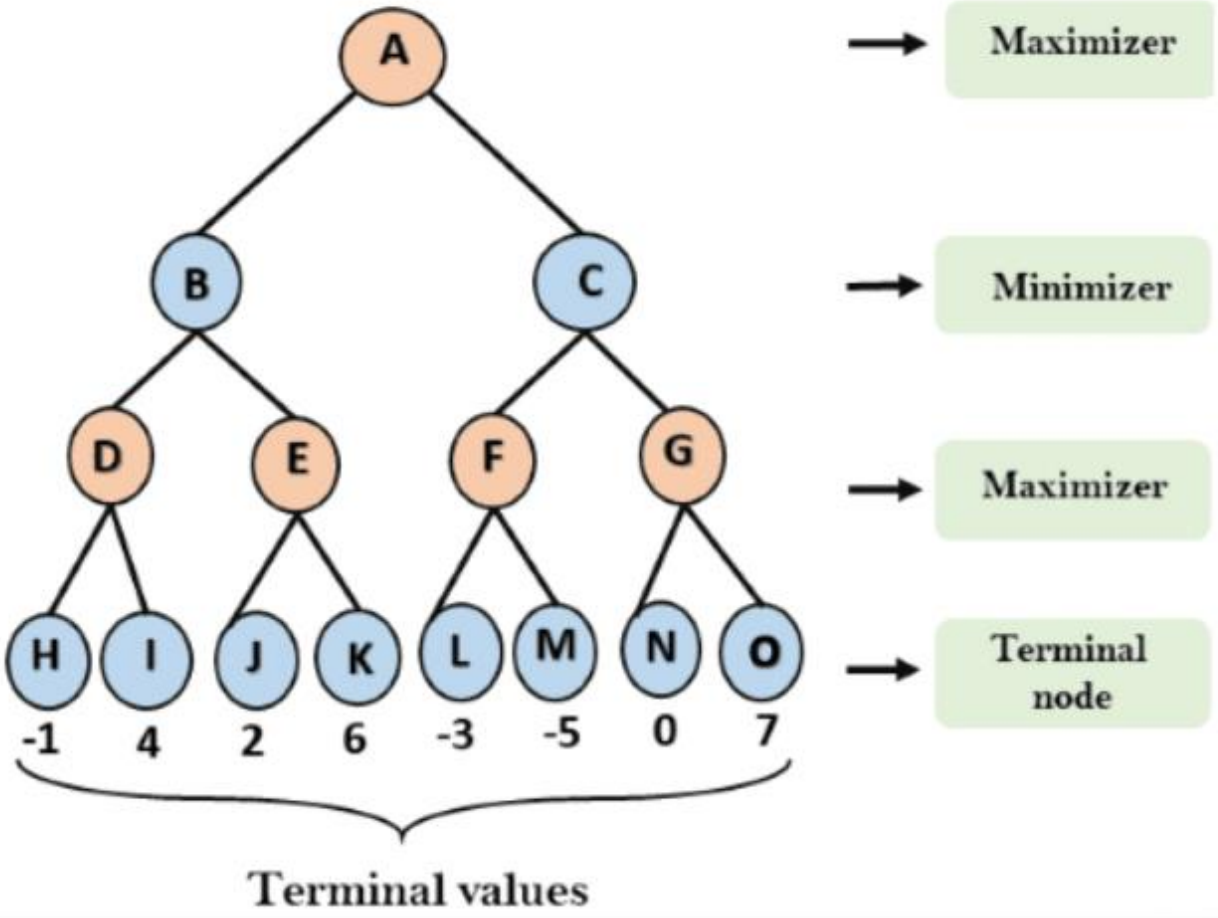


- “Initially, MAX chooses a1 and then MIN chooses b1. This guarantees a minimum gain of 3 for MAX and a maximum loss of 3 for MIN.”

- Compute MINIMAX(S) in the following game tree.

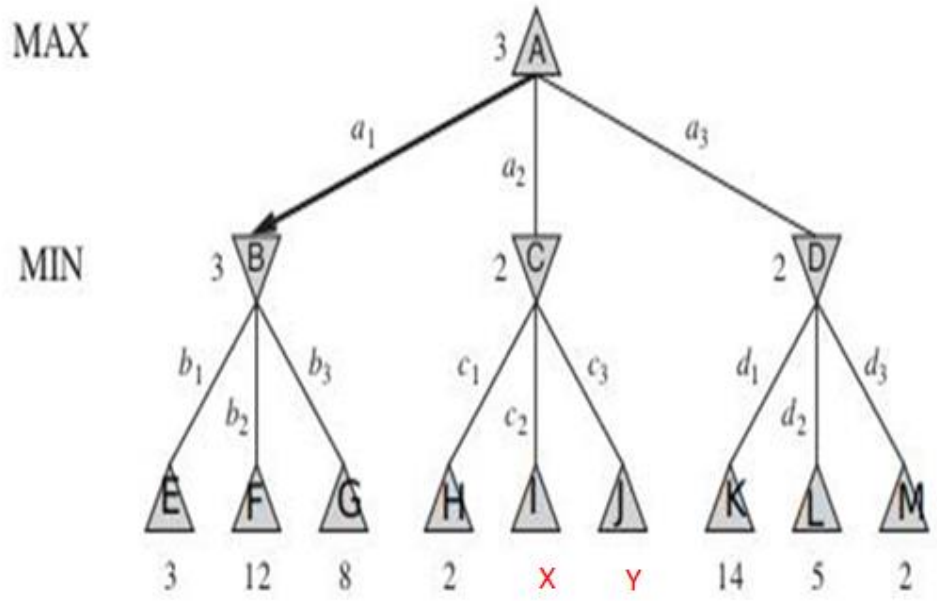


- MINIMAX(S)=Max{Min(C,D,E),Min(F,G)}
- =Max{Min(5,3,8),Min(4,2)}
- =Max{3,2}
- =3


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

MiniMax algorithm-Drawbacks

- The complexity of the MINIMAX algorithm grows exponentially with the depth of the search tree
- Inability to handle large state spaces
- Lack of consideration for opponent's strategy
- Alpha-beta pruning can be used to overcome this problem


MAX
MIN
3 A
a_1
a_2
a_3
3 B
2 C
2 D
b_1
b_2
b_3
c_1
c_2
c_3
d_1
d_2
d_3
E
F
G
H
I
J
K
L
M
3
12
8
2
X
Y
14
5
2

Alpha-beta pruning

- Alpha-beta pruning is an optimization technique for the minimax algorithm
- It helps reduce the number of nodes evaluated in the game tree, making the algorithm more efficient.

- The minimax algorithm searches the entire game tree to find the best possible move by alternating between the maximizing player (trying to maximize the score) and the minimizing player (trying to minimize the score).
- Without alpha-beta pruning, the algorithm would explore all possible moves, which can be computationally expensive, especially for games with large search spaces.
- Alpha-beta pruning works by "pruning" branches of the game tree that cannot possibly influence the final decision.
- This happens when a player has already found a better move, so it’s unnecessary to explore other moves that would not change the outcome.

- **Alpha (α):** The best value that the maximizing player can guarantee so far. Initially set to -∞
- **Beta (β):** The best value that the minimizing player can guarantee so far. Initially set to +∞.

1. **Start at the root node**: Begin at the root of the game tree with initial values $\alpha = -\infty$ (for maximizing player) and $\beta = +\infty$ (for minimizing player).
2. **Explore child nodes**: Traverse the child nodes of the current node. Alternate between maximizing and minimizing the player's turn as you go down the tree.
3. **Update α and β**:
   - If you are at a maximizing node, you update α: $\alpha = \max(\alpha,$ value of current node).
   - If you are at a minimizing node, you update β: $\beta = \min(\beta,$ value of current node).

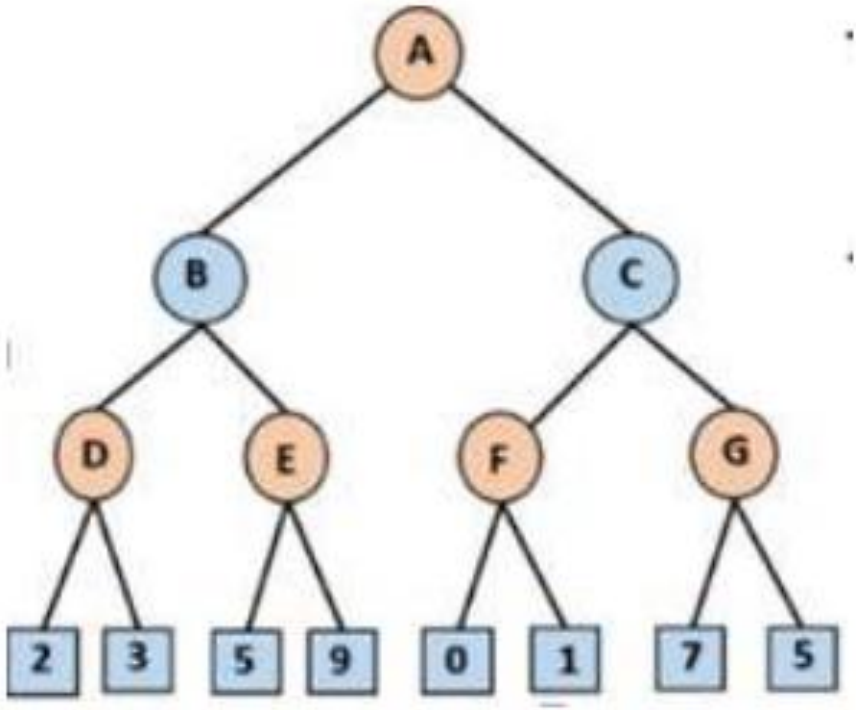
4. Prune the tree:

- If at any point, **α ≥ β**, stop evaluating that branch. This is because the opponent will never allow this branch to be chosen, so there's no point in exploring it further.
- **Pruning** happens when the current node cannot influence the result because the opponent would already choose a better option elsewhere.

5. Repeat: Continue this process recursively until the entire tree is explored or pruned.

6. Return the value of the root: Once the tree has been fully searched, the value of the root node will represent the optimal decision for the player whose turn it is.

- Determine which of the branches in the game tree below will be pruned if we apply alpha-beta pruning to solve the game (Assume that the maximising player plays first).



Advantages of using alpha beta pruning

1. **Pruning Unnecessary Branches:**

- Minimax evaluates all possible moves. Alpha-beta pruning skips over branches of the search tree that will not affect the final decision.

2. Increased Efficiency (Time Complexity):

- Minimax: Without pruning, the time complexity of the algorithm is **O(b^d),** where b is the branching factor and d is the depth of the tree.
- Alpha-beta pruning: The time complexity is **O(b^(d/2))** in the best case, which can be a significant improvement, especially for deep search trees.