UNIT 5

INT426:GENERATIVE ARTIFICIAL INTELLIGENCE

Search Algorithms in Games

- Search algorithms are used in games to figure out a strategy.
- The algorithms search through the possibilities and pick the best move.
- There are various parameters to think about – speed, accuracy, complexity, and so on.
- The goal of these algorithms is to find the optimal set of moves that will help them arrive at the final condition.
- Every game has a different set of winning conditions. These algorithms use those conditions to find the set of moves.

Role of Combinatorial search

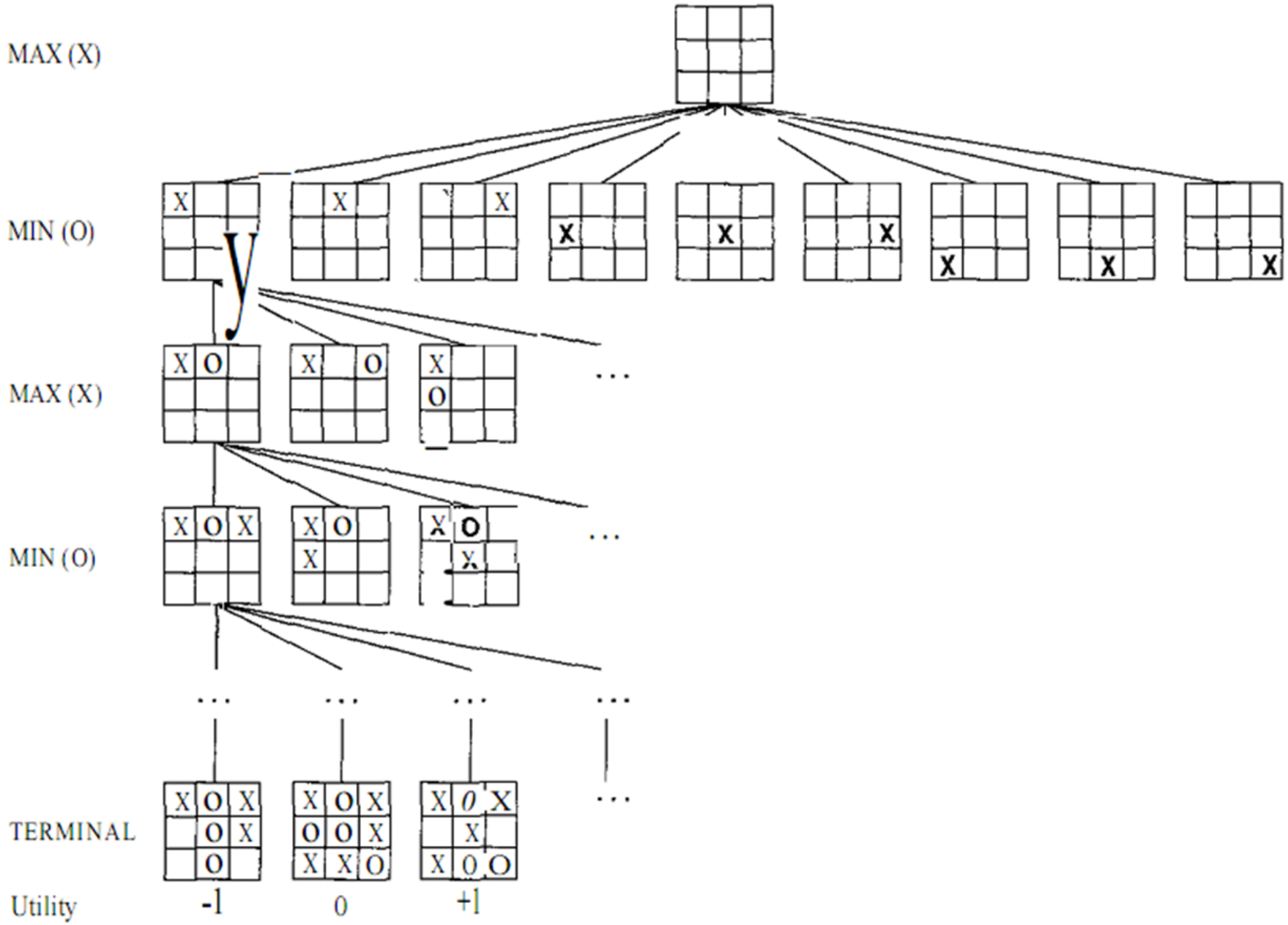
- Search algorithms appear to solve the problem of adding intelligence to games, but there's a drawback. These algorithms employ a type of search called exhaustive search, which is also known as ***brute force search***.
- It basically explores the entire search space and tests every possible solution.
- It means that, in the worst case, we will have to explore all the possible solutions before we get the right solution.

- As the games get more complex, we cannot rely on brute force search because the number of possibilities gets enormous. This quickly becomes computationally intractable.
- It refers to a field of study where search algorithms efficiently explore the solution space using **heuristics or by reducing the size of the search space**. This is very useful in games like Chess or Go.

- ***Combinatorial search works*** *efficiently by using* ***pruning strategies****. These strategies help it* ***avoid testing all possible solutions by eliminating the ones that are obviously wrong****. This helps save time and effort.*

- **A game can be formally defined as a kind of search problem with the following components:**

- **The initial state,** which includes the board position and an indication of whose move it is.
- **A set** of **operators,** which define the legal moves that a player can make.
- A **terminal test,** which determines when the game is over. States where the game has ended are called **terminal states.**
- A **utility function** (also called a **payoff function),** which gives a numeric value for the outcome of a game. In chess, the outcome is a win, loss, or draw, which we can represent by the values +1, —1, or 0. Some games have a wider variety of possible outcomes; for example, the payoffs in backgammon range from +192 to —192.


MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
-1
0
+1

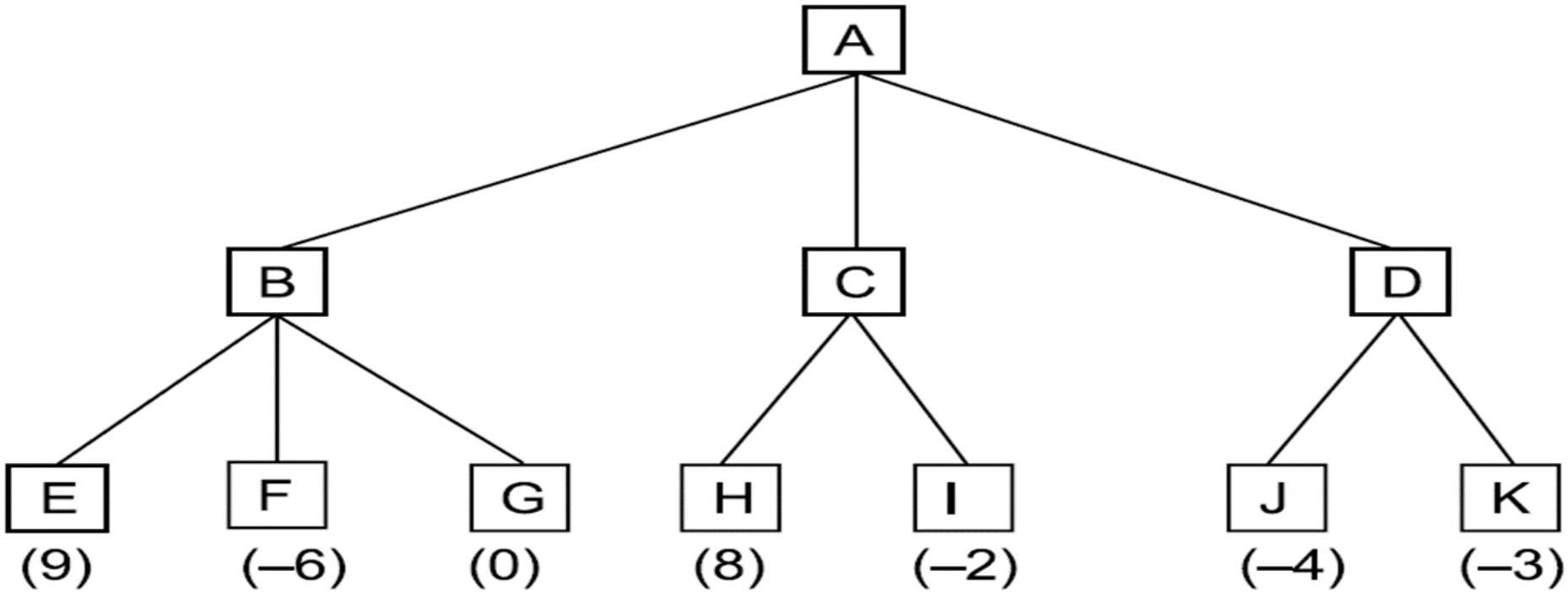
Minimax Algorithm

- The heuristics are used to speed up the search strategy and the Minimax algorithm is one such strategy used by combinatorial search.
- When two players are playing against each other, they are basically working towards opposite goals. So each side needs to predict what the opposing player is going to do in order to win the game. Keeping this in mind, **Minimax tries to achieve this through strategy**.
- It will try to minimize the function that the opponent is trying to maximize.

- Minimax 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.
- MinMax algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-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**. where **MAX will select the maximized** value and **MIN will select the minimized value**.

- The **minimax** algorithm performs a **depth-first search algorithm** for the exploration of the complete game tree.
- **The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.**

Two – Ply Search



```
function minimax(position, depth, maximizingPlayer)
    if depth == 0 or game over in position
        return static evaluation of position

    if maximizingPlayer
        maxEval = -infinity
        for each child of position
            eval = minimax(child, depth - 1, false)
            maxEval = max(maxEval, eval)
        return maxEval

    else
        minEval = +infinity
        for each child of position
            eval = minimax(child, depth - 1, true)
            minEval = min(minEval, eval)
        return minEval
```

Properties of Mini-Max algorithm:

Complete- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
Optimal- Min-Max algorithm is optimal if both opponents are playing optimally.
Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(b^m), where b is branching factor of the game-tree, and m is the maximum depth of the tree.
Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

Alpha-Beta pruning

- It is a procedure to prune-off (ignore) irrelevant nodes of the game tree to be traversed.
- Pruning does not affect the final result and gets the exact same result as does full minimax.
- It can speed up a depth-first minimax search.
- Alpha (α): A lower bound on the value that a max node may ultimately be assigned.
- Beta (β): An upper bound on the value that a minimizing node may ultimately be assigned.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

$$\alpha >= \beta$$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Why α-β Pruning???

- Time complexity of conventional minimax algorithm is $O(b^d)$.

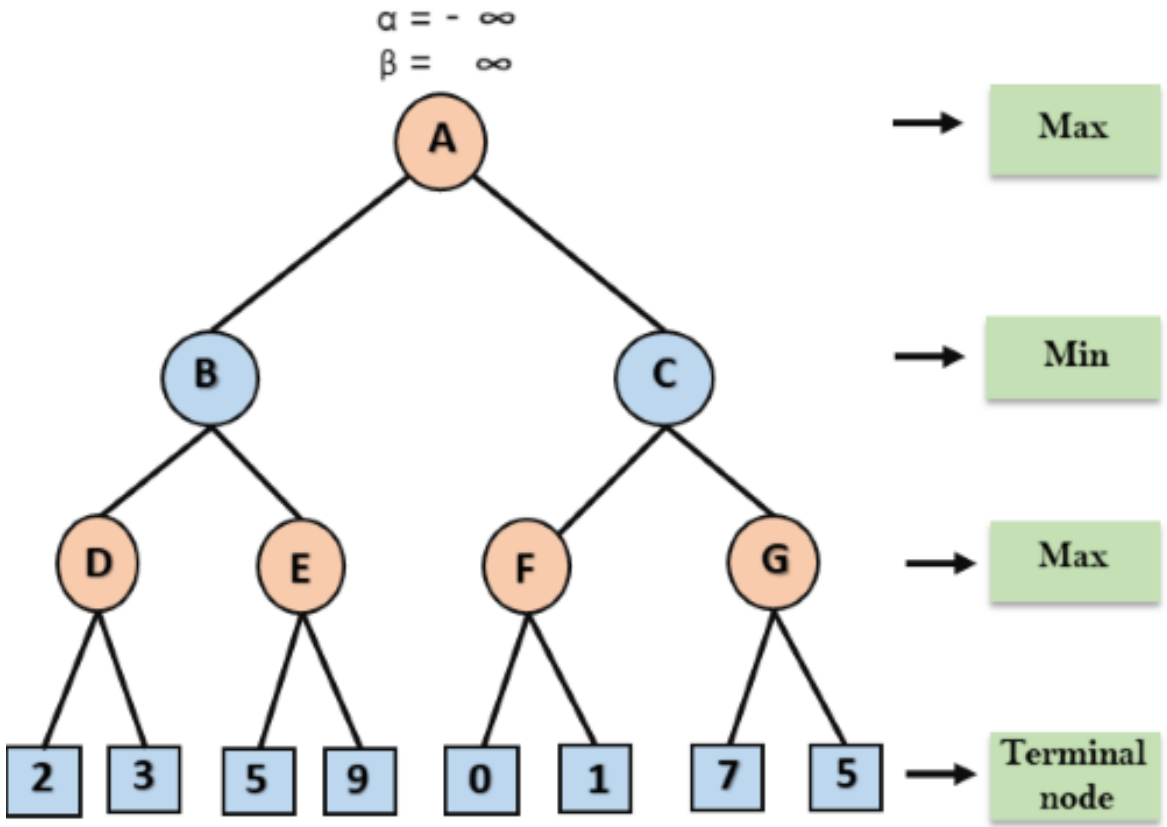
  b: branch factor

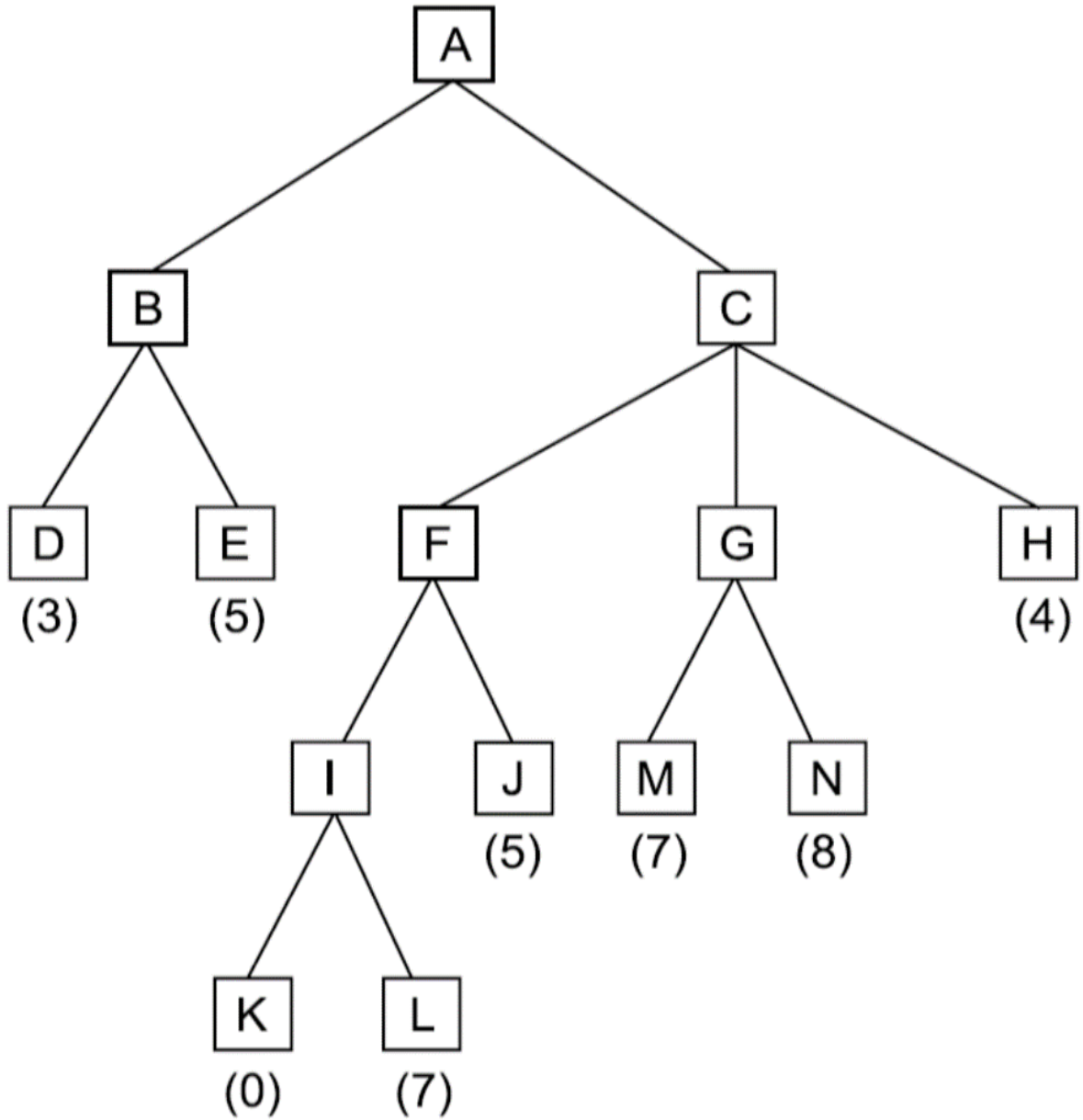
  d: depth

- Chess: $O(35^{100})$

- By pruning-off the irrelevant nodes, this complexity can be reduced to $O(b^{d/2})$.

Examples





Maximizing ply

Minimizing ply

Maximizing ply

Minimizing ply

Pseudo-code for Alpha-beta Pruning:

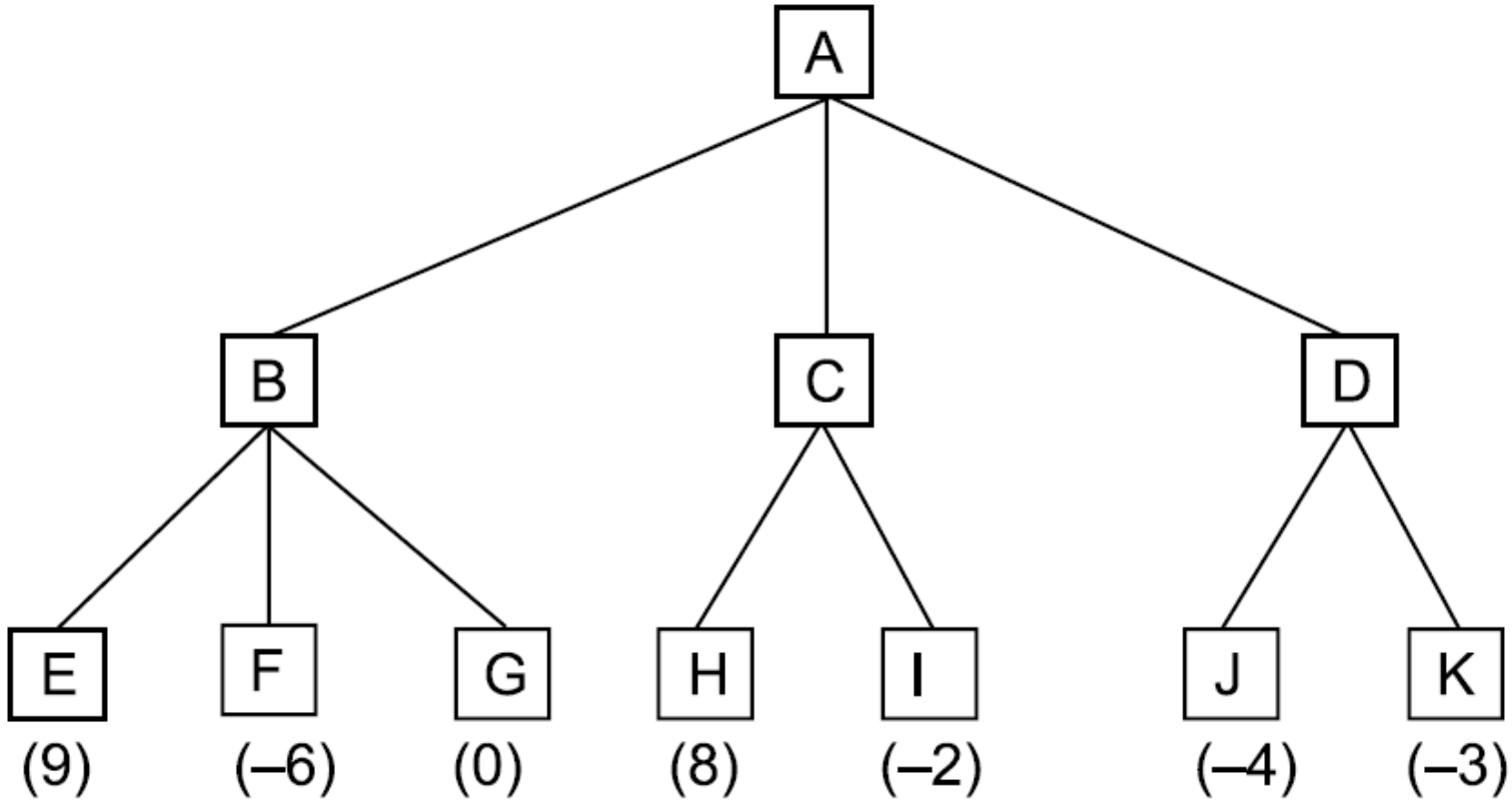
```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
   maxEva= -infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, False)
  maxEva= max(maxEva, eva)
  alpha= max(alpha, maxEva)
   if beta<=alpha
 break
 return maxEva

else                    // for Minimizer player
   minEva= +infinity
   for each child of node do
   eva= minimax(child, depth-1, alpha, beta, true)
   minEva= min(minEva, eva)
   beta= min(beta, eva)
    if beta<=alpha
  break
 return minEva
```

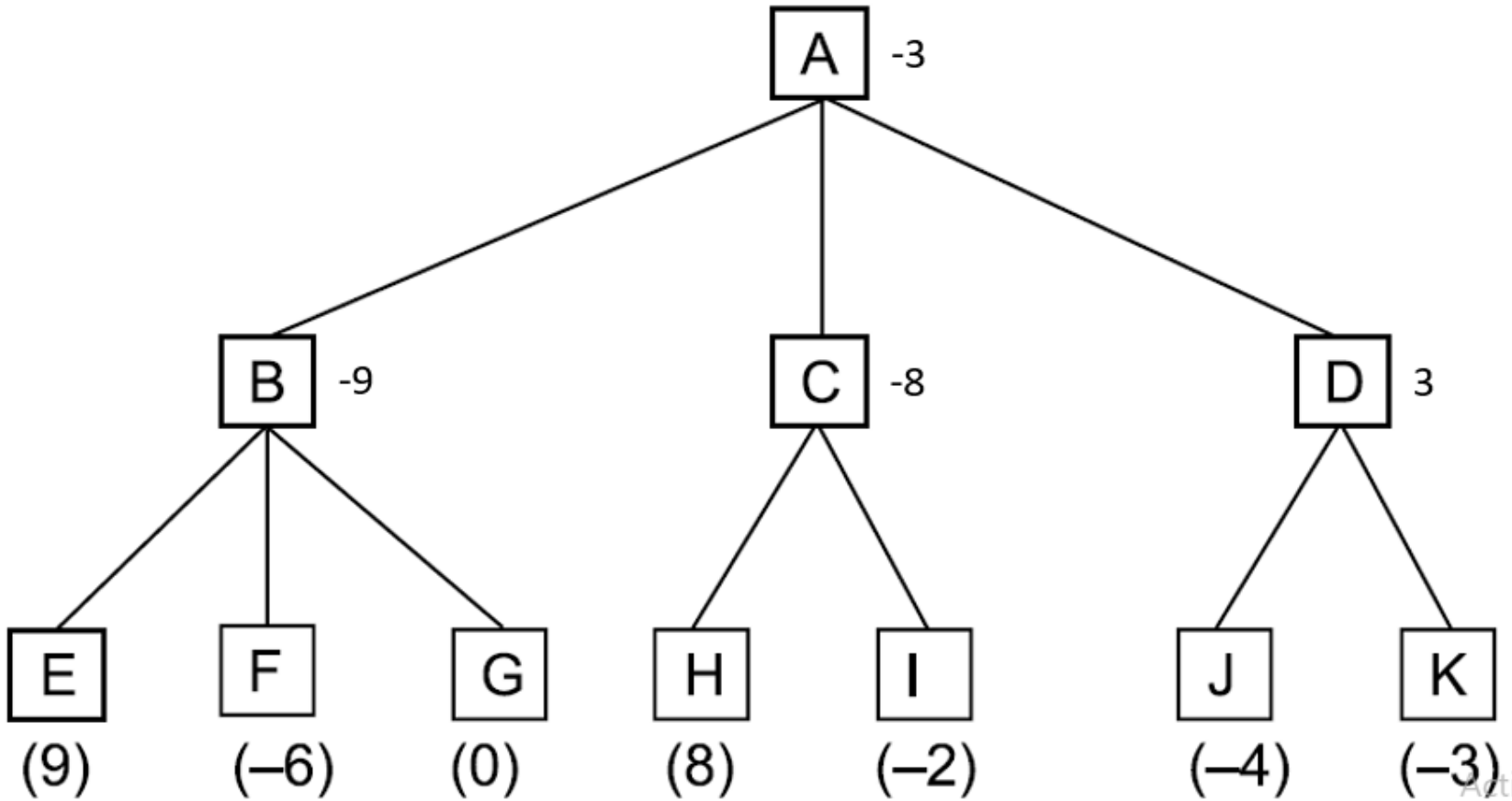
Negamax Algorithm

Introduction

- It is a variant of Minimax.
- It is frequently used in real world implementations.
- A two-player game is usually a zero-sum game.
- One player's loss is equal to another player's gain and vice versa.
- Negamax uses this property extensively to come up with a strategy to increases its chances of winning the game.

- In terms of the game, the value of a given position to the first player is the negation of the value to the second player.
- Each player looks for a move that will maximize the damage to the opponent.
- The value resulting from the move should be such that the opponent gets the least value.
- This works both ways seamlessly, which means that a single method can be used to value the positions.


A
B
C
D
E
F
G
H
I
J
K
(9)
(–6)
(0)
(8)
(–2)
(–4)
(–3)


A
-3
B
-9
C
-8
D
3
E
F
G
H
I
J
K
(9)
(–6)
(0)
(8)
(–2)
(–4)
(–3)