



# Game Playing

# Game Playing

- Game playing is a **Major topic** of AI. It requires **intelligence**, and has certain well defined **states and rules**.
- A Game is defined as a **sequence of choices** where each choice is from a number of discrete alternatives.
- Each sequence ends in a certain outcome and has a **definite value** for the player.
- Human experts playing against computers has always been a great fashion.
- Games are classified into:
- **Perfect information games**: Both the players have access to same information. ex. chess, checker etc.
- **Imperfect Perfect information games**: The players don't have access to complete information. ex. Cards, Dice

- A game is **discrete** if it has finite number of states.
- A typical characteristic is to **look ahead** at future positions in order to succeed.
- 
- **Optimal solution** can be achieved by **exhaustive search** technique if there are **no constraints** on **time and space**.
- But for most of the games the solution is **inefficient** using exhaustive search technique .

# Game problem vs State space problem

**Comparisons between State Space Problems and Game Problems**

<b>State Space Problems</b>	<b>Game Problems</b>
States	Legal board positions
Rules	Legal moves
Goal	Winning positions

- A game begins from a specified initial state and ends in a position can be declared as **WIN, LOSS and DRAW**.
- A **Game tree** is an explicit information of all possible plays of the game.
- Each path from root to a terminal node represents a **complete path** of the game.
- Game theory is based on the philosophy of **minimizing the maximum loss and maximizing the minimum gain**.
- Max player is **computer** and Min player is **human**.
- Our aim is to make the **computer** to win by always taking best possible move at its turn.

# Status Labelling Procedure in Game tree

- If  $j$  is a non-terminal MAX node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN,} & \text{if any of } j\text{'s successor is a WIN} \\ \text{LOSS,} & \text{if all } j\text{'s successor are LOSS} \\ \text{DRAW,} & \text{if any of } j\text{'s successor is a DRAW and is WIN} \end{cases}$$

- If  $j$  is a non-terminal MIN node, then

$$\text{STATUS}(j) = \begin{cases} \text{WIN,} & \text{if all } j\text{'s successor are WIN} \\ \text{LOSS,} & \text{if any of } j\text{'s successor is a LOSS} \\ \text{DRAW,} & \text{if any of } j\text{'s successor is a DRAW and none is LOSS} \end{cases}$$



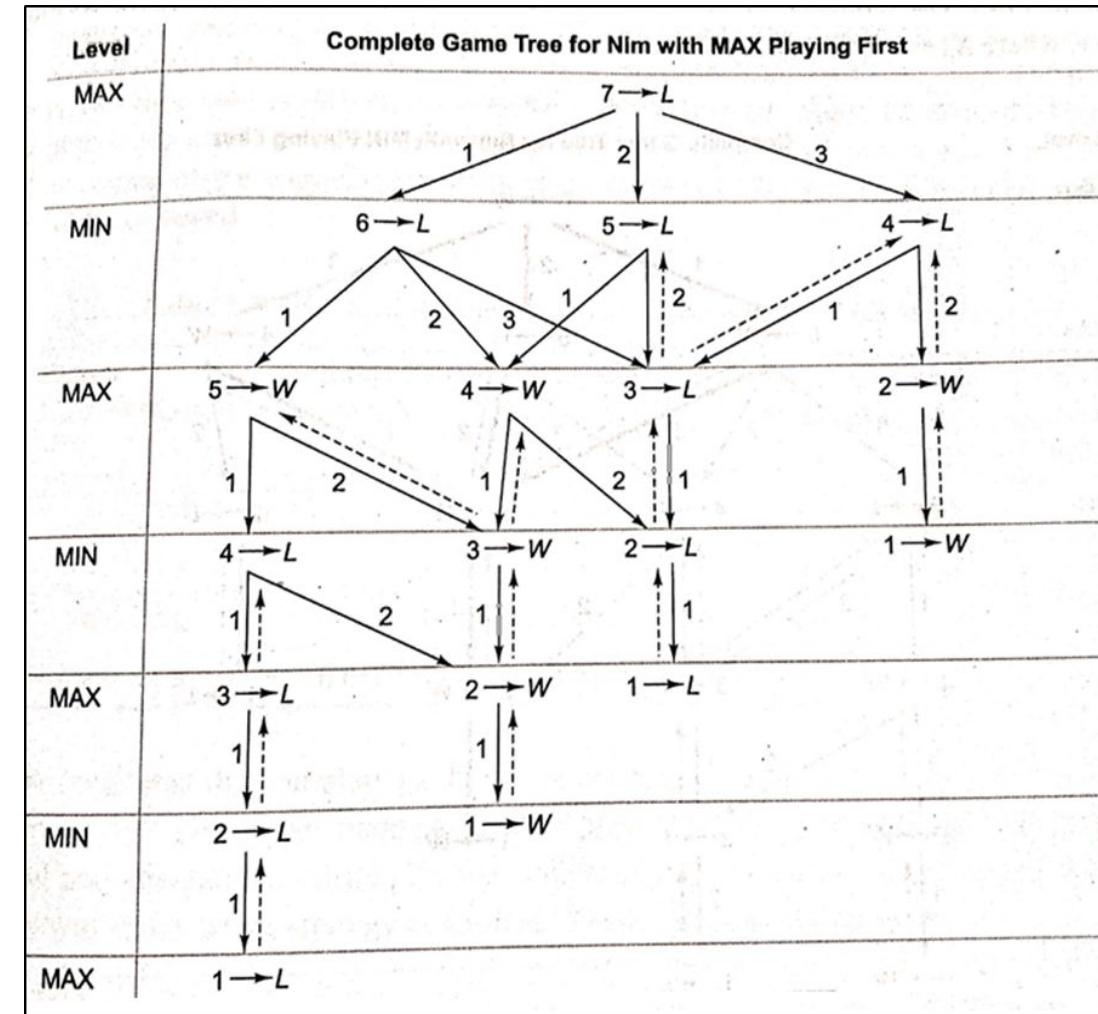
# Nim Game Problem

- There is a single pile of matchsticks ( $>1$ ) and two players
- Moves are made by the player alternately
- In each move, each player can pick up a maximum of half the number of matchsticks in the pile.
- Whoever picks the last matchstick loses.



# Nim Game Problem

- Mark status of terminal
- Mark the status of non terminal by using status labelling procedure
- Root ( $\text{MAX} \rightarrow L$ )
- **MIN always wins** irrespective of the move made by the first player

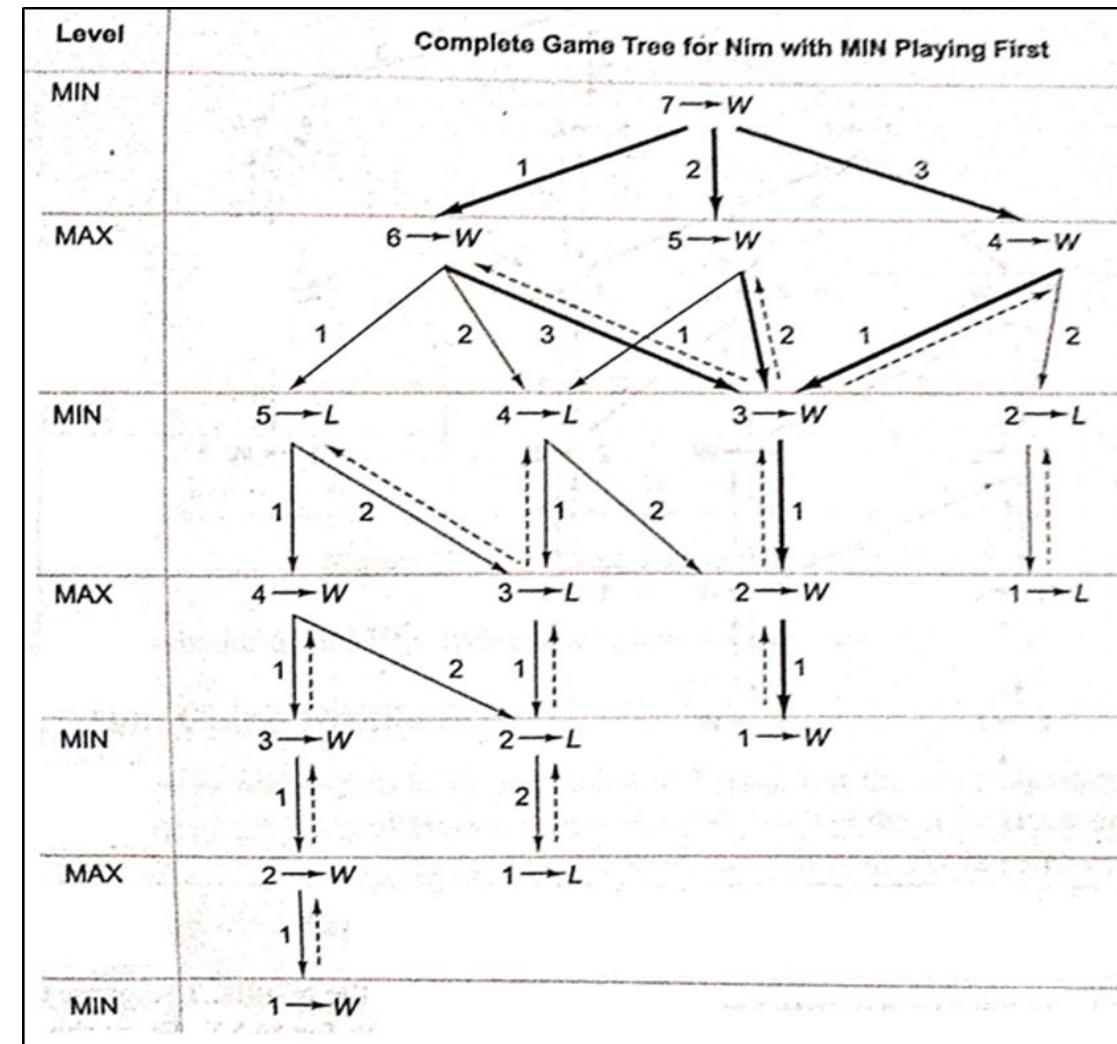


Game Tree for Nim in which MAX Plays First



# Nim Game Problem

- Mark status of terminal
- Mark the status of non terminal by using status labelling procedure
- Root (**MIN**→W)
- **MIN always wins** irrespective of the move made by the first player



**Game Tree for Nim in which MIN Plays First**



# Strategy

- If at the time of MAX player's turn there are N matchsticks in a pile ,

- then Max can force a win by leaving M matchsticks for the MIN player to play , where

$$M \in \{1,3,7,15,31,63,\dots\}$$

- using the rule of game (that is MAX can pick up a maximum of half the number of matchsticks in the pile).

- The sequence  $\{1,3,7,15,31,63,\dots\}$  can be generated using the formula

$$X_i = 2 X_{i-1} + 1 \text{ for } i > 0 \text{ where } X_0 = 1$$



# Strategy

- First method

- Look up from the sequence {1,3,7,15,31,63,...} and figure out the closest number less than the given number N of match sticks in the pile .
- The difference between N and the number gives the desired number of sticks that have to be picked up .
- For example , if N=45, the closest number to 45 in the sequence is 31, so we obtain the desired number of matchsticks to be picked up as 14 on subtracting 31 from 45.
- This way try to maintain a sequence {1,3,7,15,31,63,...}.

- Second method

- the desired number is obtained by removing the most significant digit from the binary representation of N and adding it to the least significant digit position.

N	Binary Representation	Sum of 1 with MSD removed from N	Number of sticks to be removed	Number left in pile
13	1101	0101+0001	0110=6	7
27	11011	01011+00001	01100=12	15
36	100100	000100+000001	000101=5	31
70	1000110	0000110+0000001	0000111=7	63



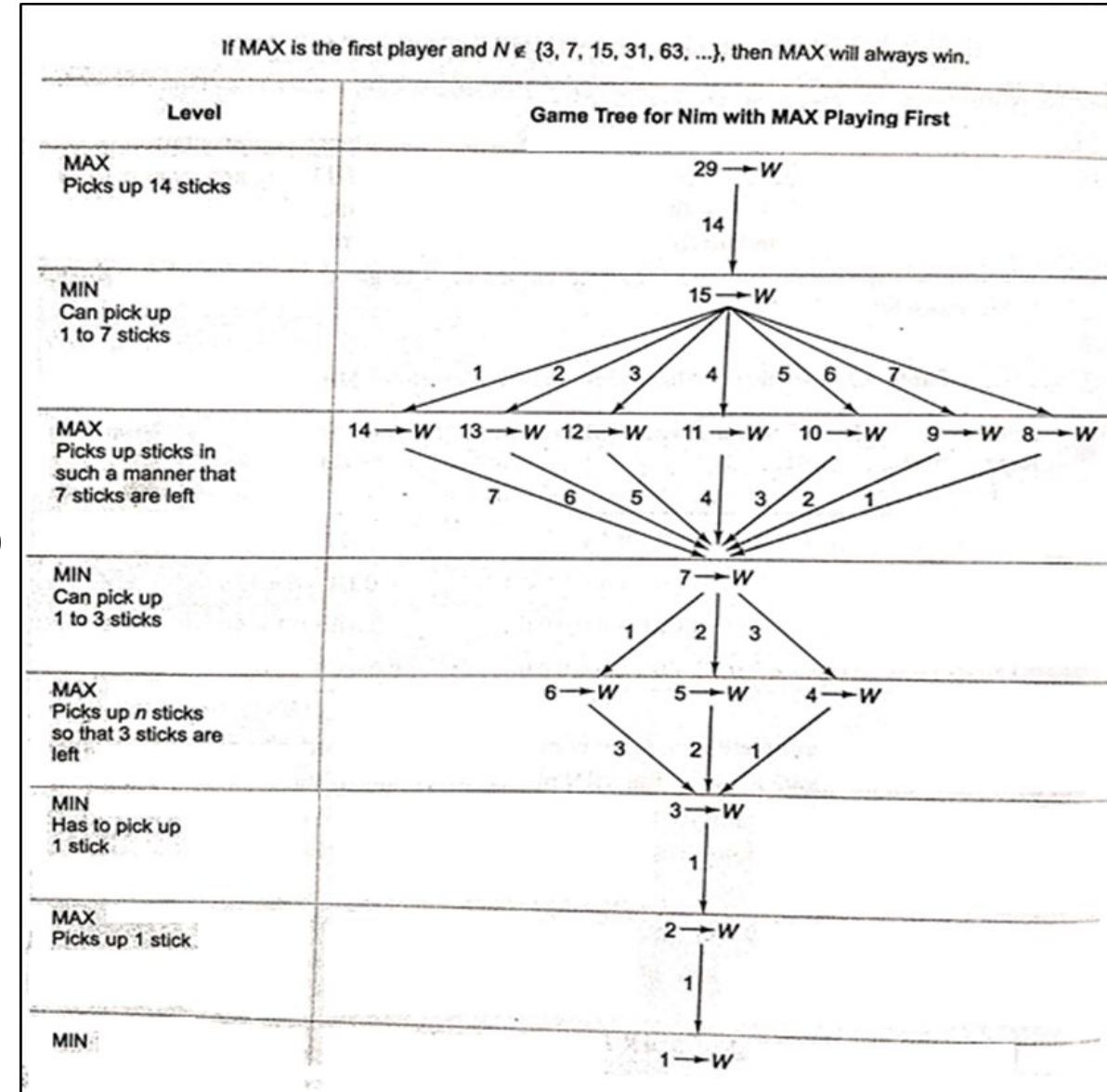
# MAX - WIN

- We can formulate two cases where MAX will always win if the following strategy is applied.
  - **CASE 1** MAX is the **first player** and initially there are  $N \notin \{3, 7, 15, 31, 63, \dots\}$  matchsticks.
  - **CASE 2** MAX is the **second player** and initially there are  $N \in \{3, 7, 15, 31, 63, \dots\}$  matchsticks.



## Validity of Cases for Winning of MAX Player

- **CASE 1** If MAX is the first player and  $N \notin \{3, 7, 15, 31, 63, \dots\}$ .
- MAX will always win.
- Consider a pile of 29 sticks and let MAX be the first player.

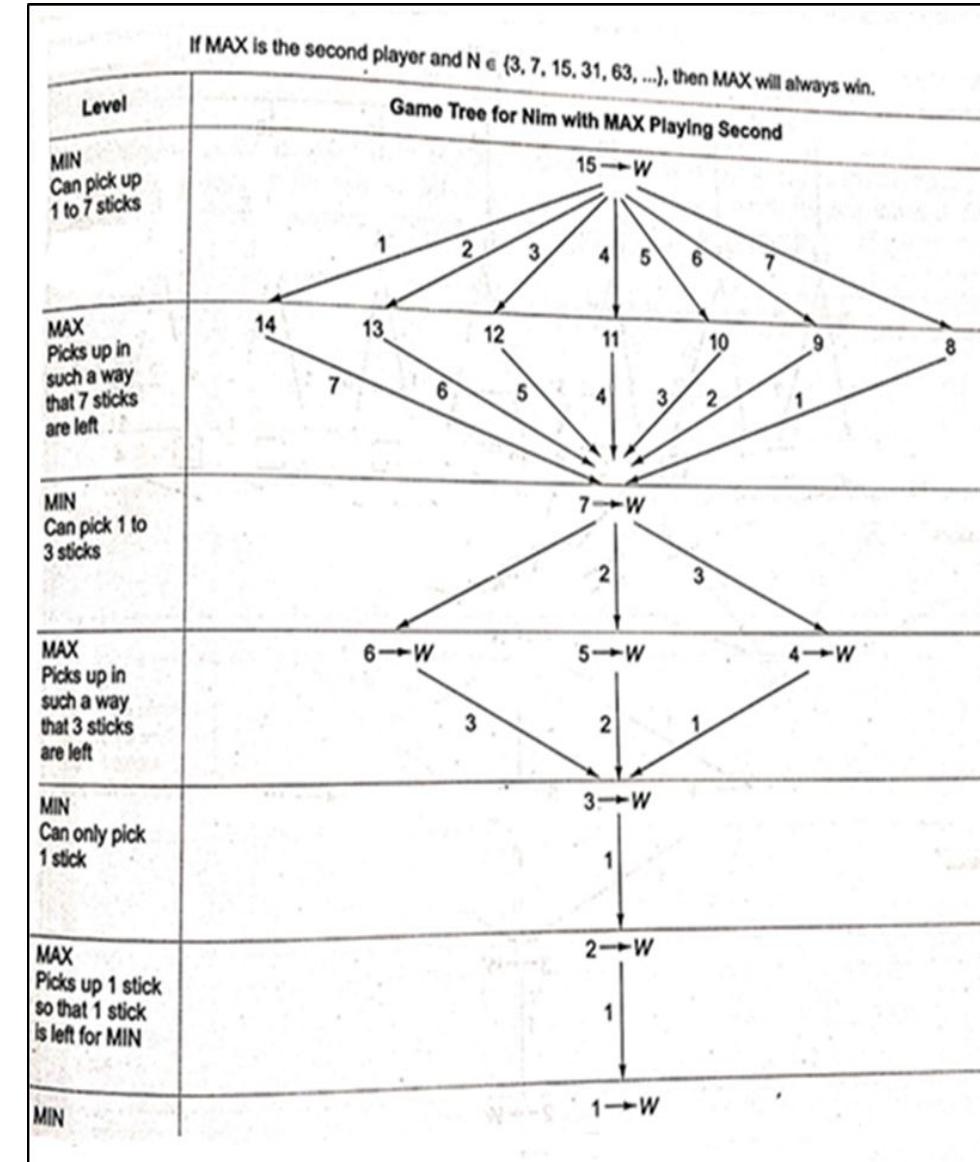


Validity of Case 1 (Example for  $N = 29$ )



## Validity of Cases for Winning of MAX Player

- **CASE 2** If MAX is the second player and  $N \in \{3, 7, 15, 31, 63, \dots\}$
- Consider a pile of 15 sticks and let MAX be the second player.
- The complete game tree for this case is shown in fig.

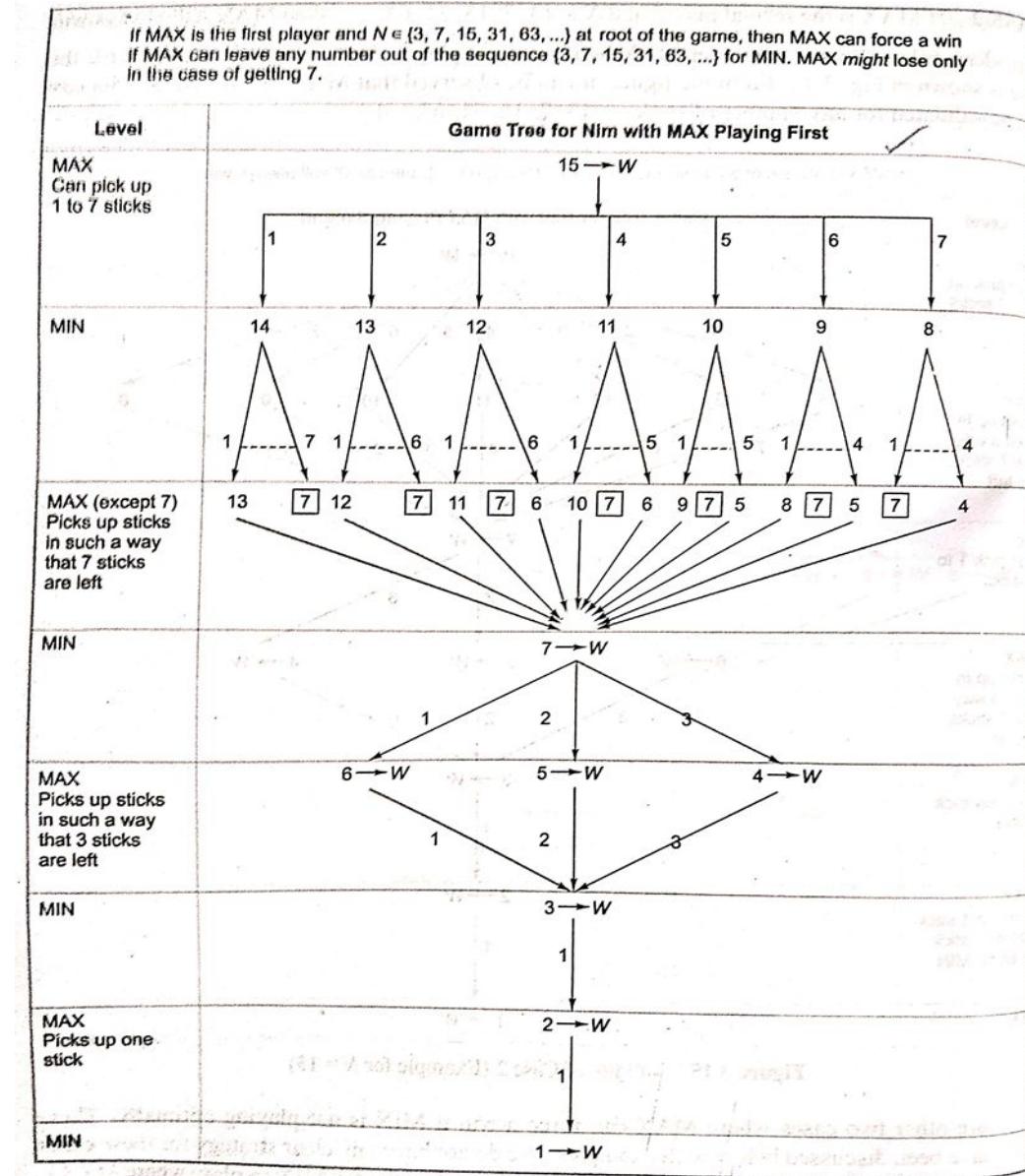


Validity of Case 2 (Example for  $N = 15$ )



## Validity of Cases for Winning of MAX Player

- **CASE 3** If MAX is the first player and  $N \in \{3, 7, 15, 31, 63, \dots\}$  at root of the game,
- MAX can force a win using the strategy mentioned above in all cases except when MAX gets a number from the sequence  $\{3, 7, 15, 31, 63, \dots\}$  at its turn.
- Assume that  $N=15$  fig 1 shows that MAX wins in all cases **except when it gets 7 matchsticks in its turn.**
- MAX can even win the game when it gets 7 sticks at its turn in the game for all values except when it gets 3 sticks at its turn in the game.

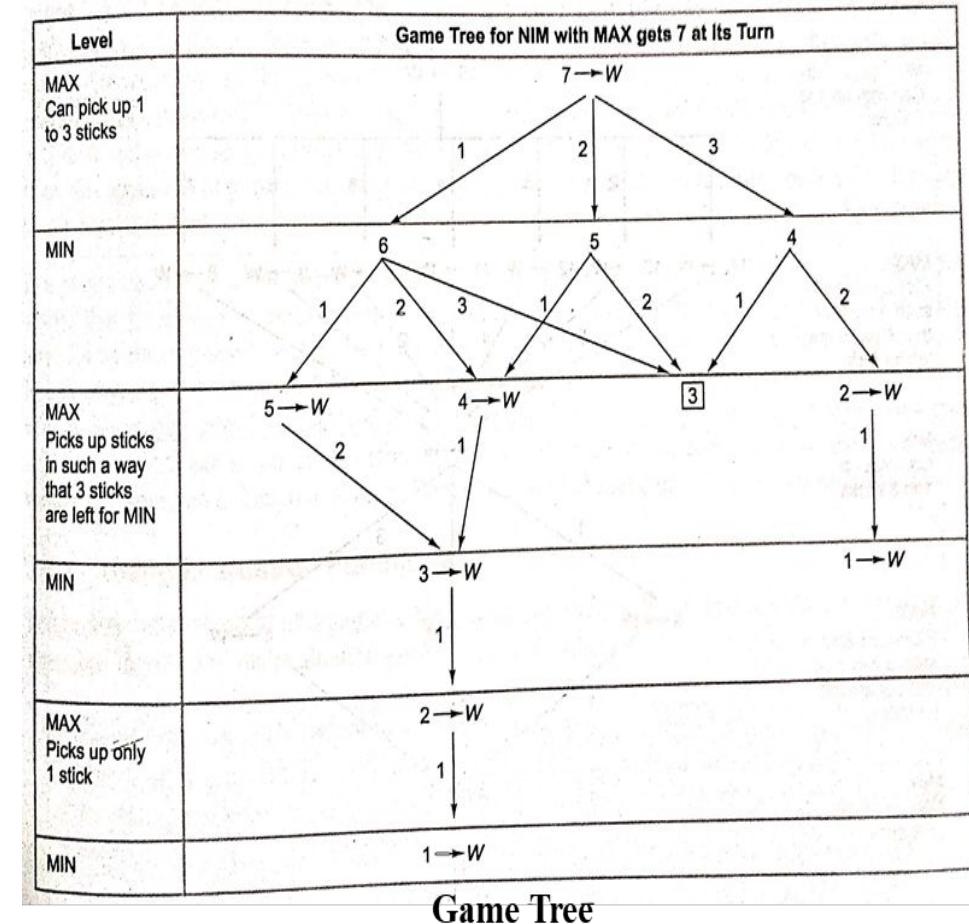


Validity of Case 3 (Example for  $N = 15$ )



## Validity of Cases for Winning of MAX Player

- MAX can even win the game when it gets 7 sticks at its turn in the game for all values except when it gets 3 sticks at its turn in the game.

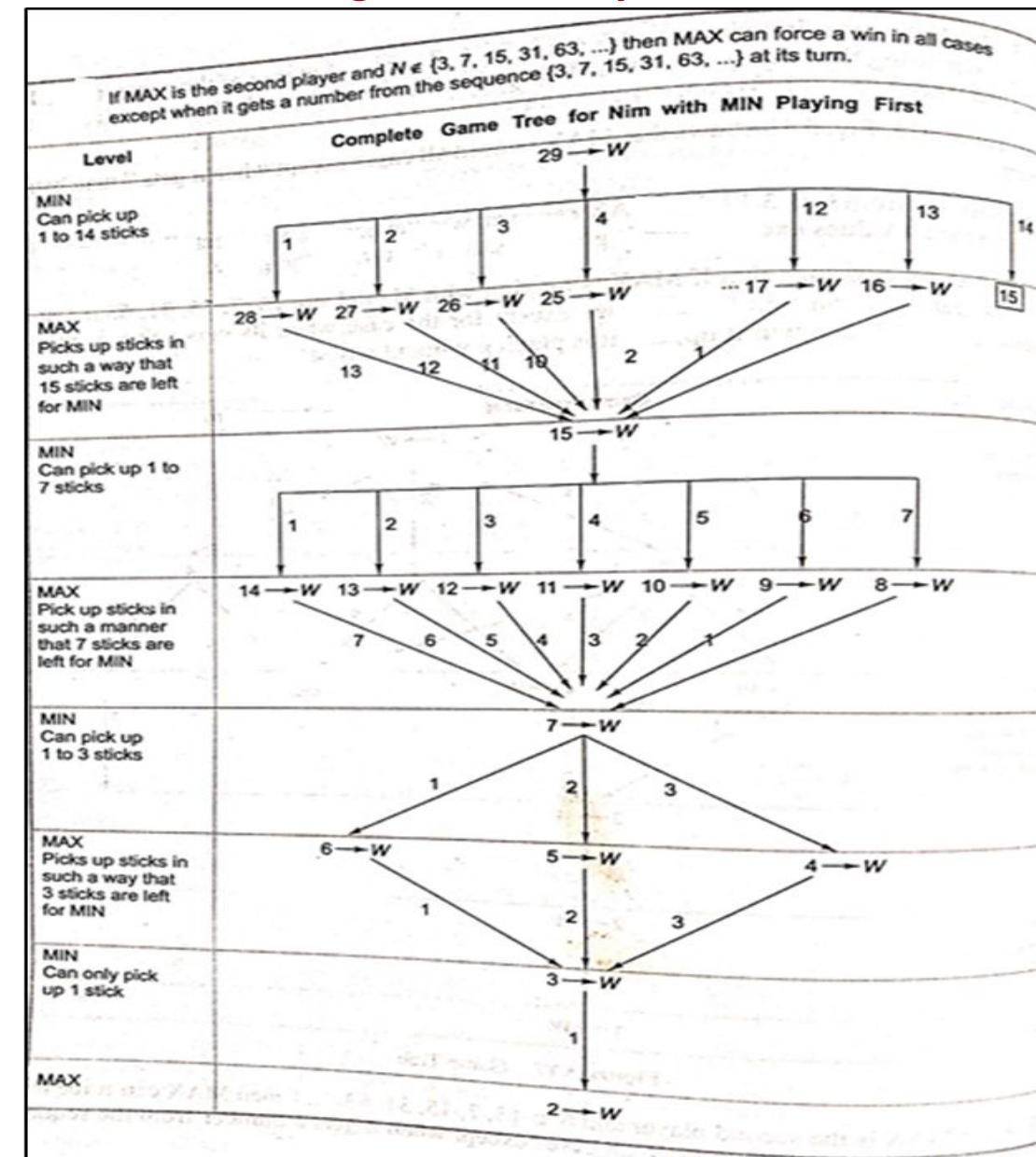




## Validity of Cases for Winning of MAX Player

- **CASE 4** If MAX is the second player and  $N \notin \{3, 7, 15, 31, 63, \dots\}$  then MAX can force a win using the above mentioned strategy in all cases except when it gets a number from the sequence  $\{3, 7, 15, 31, 63, \dots\}$  at its turn.

- Let  $N=29$
- Let MIN be the first player.
- MAX wins in all cases except when it gets 15 matchsticks at its turn.
- Max might lose in case of it getting 15.



Validity of Case 4 (Example for  $N = 29$ )



## Bounded Look-Ahead Strategy and use of Evaluation Functions

- Status labelling procedure requires the generation of complete tree or atleast some part of it
- For many games in reality too large trees may be generated & need to be evaluated backward from terminal to root in order to determine the optimal first move.
- This approach may not be practical
- It is better to look for about few levels before deciding the move
- **Look-ahead strategy** : The player can develop the game tree to some extent before deciding on the move.
- **N-move Look-ahead strategy** : The player is looking ahead n no of levels before making a move.



# Bounded Look-Ahead Strategy and use of Evaluation Functions

## Using Evaluation Functions

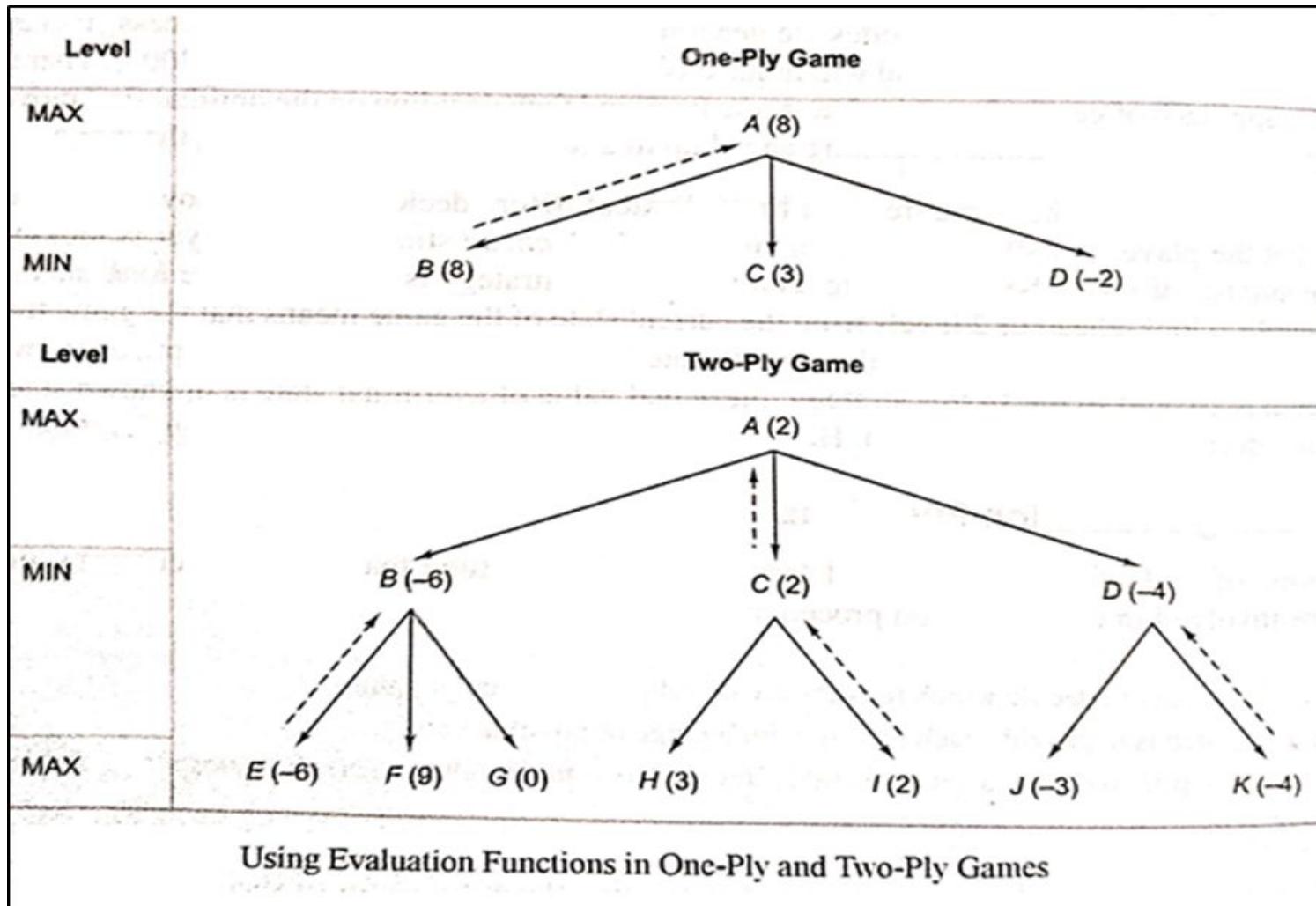
- The **first step** is to decide which features are of value in a particular game.
- The **next step** is to provide each feature with a **range of possible values**.
- The **last step** is to **devise a set of weights** in order to combine all the feature values into a single value.

## Alternative is using heuristic approximation.

- A proper **static evaluation function** can convert all judgments about board situations into a **single overall quality number**
- **Function provides numerical assessment** which indicates the favorable condition for MAX.
- **+ve value  $\Rightarrow$  good** and **-ve value  $\Rightarrow$  bad** for MAX.
- The heuristic value of nodes are determined at some level (one-ply /two-ply).
- The procedure through which the scoring information travels up the game is called **MINIMAX** procedure



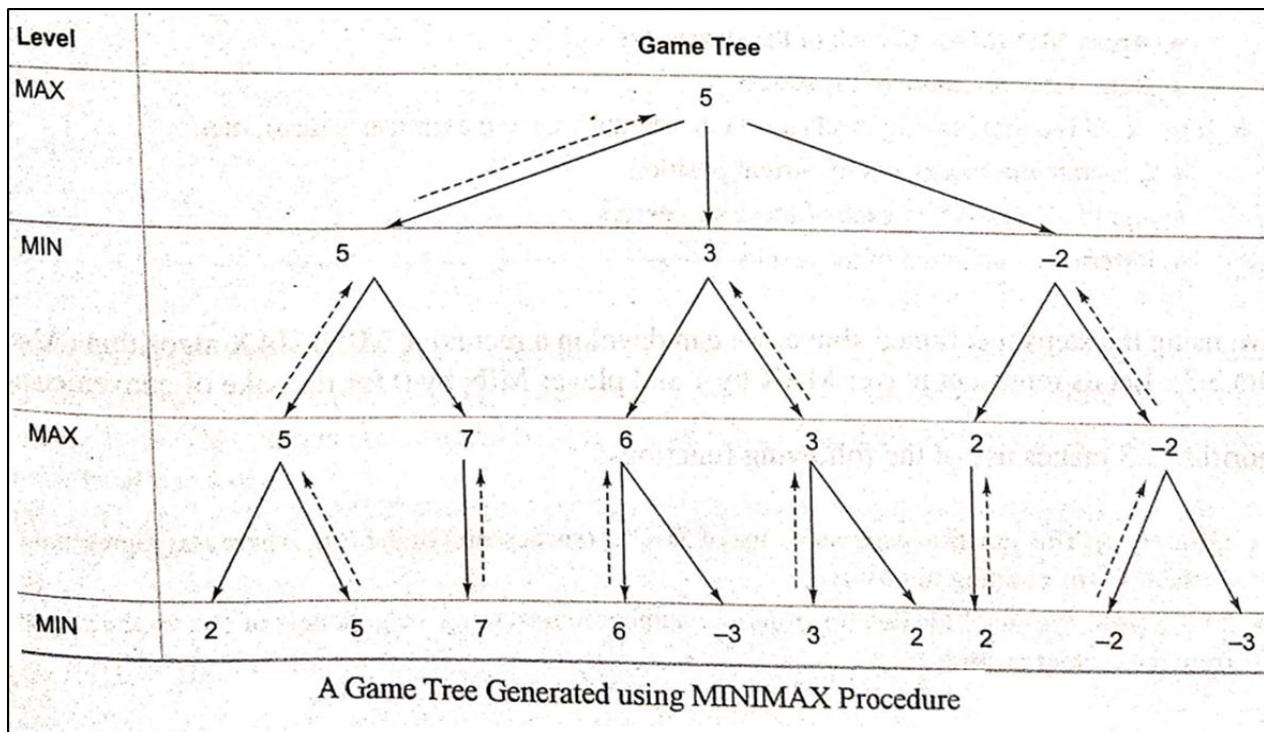
# One-Ply & two-Ply





# MINIMAX Procedure

- It is the procedure through which the scoring info travels up the game tree.
- Propagates to root, the estimated score generated by heuristic evaluation function.
- It is recursive algorithm.
- Player tries to maximize its chance of a win while minimizing that of the opponent.
- A player with +ve number is maximizing player
- A player with -ve number is minimizing player
- At each move MAX player tries to take a path that leads to a large +ve number while opponent will try to face the game towards -ve static evaluation.





## MINIMAX Procedure

The algorithmic steps of this procedure may be written as:

- Keep on generating the search tree till the limit, say depth  $d$  of the tree, has been reached from the current position.
- Compute the static value of the leaf nodes at depth  $d$  from the current position of the game tree using evaluation function.
- Propagate the values till the current position based on MINIMAX strategy.



## MINIMAX Strategy

Steps in MINIMAX strategy are written as follows:

- If the **level is minimizing** level(level reached during minimizer's turn),then

1. Generate the successors of the current position

2. Apply MINIMAX to each of the successors

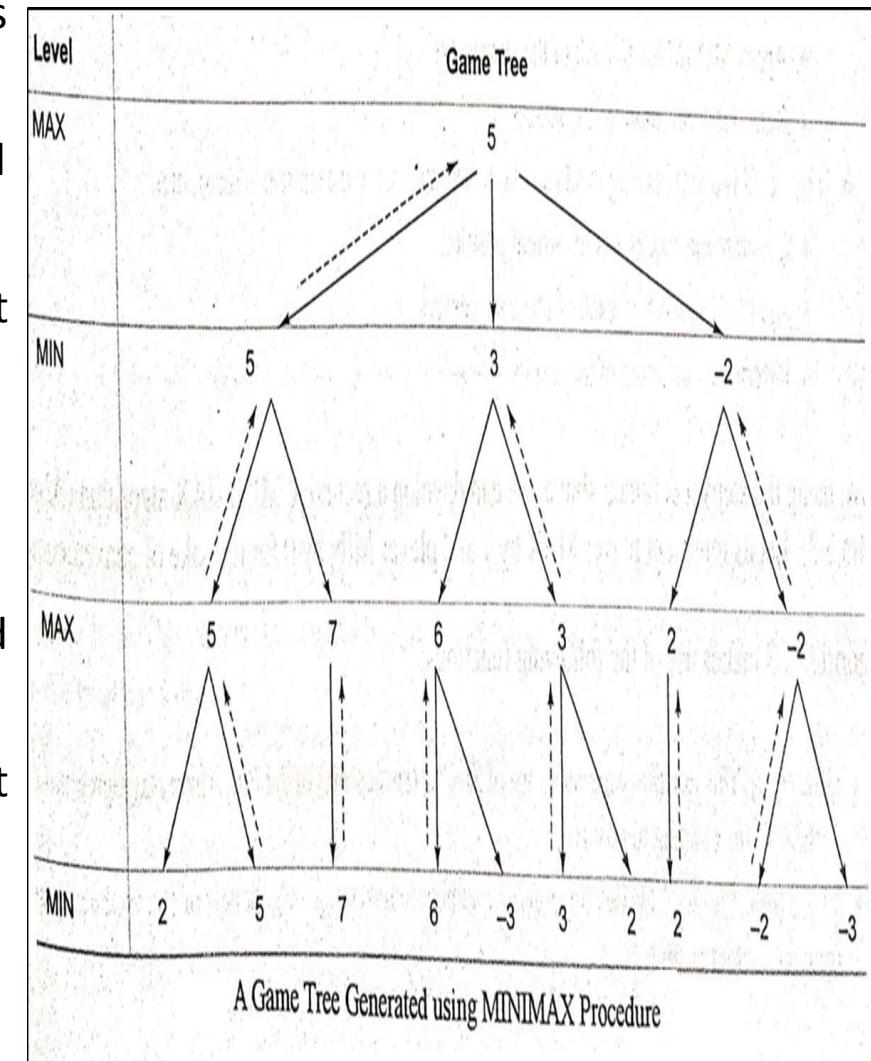
3. Return the **minimum** of the results

- If the **level is maximizing** level(level reached during maximizer's turn),then

1. Generate the successors of the current position.

2. Apply MINIMAX to each of the successors

3. Return the **maximum** of the results



```
1.function minimax(node, depth, maximizingPlayer) is
2.if depth ==0 or node is a terminal node then
3.return static evaluation of node
4.
5.if MaximizingPlayer then // for Maximizer Player
6.maxEva= -infinity
7. for each child of node do
8. eva= minimax(child, depth-1, false)
9.maxEva= max(maxEva,eva) //gives Maximum of the values
10.return maxEva
11.
12.else // for Minimizer player
13. minEva= +infinity
14. for each child of node do
15. eva= minimax(child, depth-1, true)
16. minEva= min(minEva, eva) //gives minimum of the values
17. return minEva
```

```
function MINIMAX-DECISION(state) returns an action
    return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(s, a))$ 
```

---

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow -\infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$ 
    return v
```

---

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
     $v \leftarrow \infty$ 
    for each a in ACTIONS(state) do
         $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$ 
    return v
```

**Figure 5.3** An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation  $\arg\max_{a \in S} f(a)$  computes the element *a* of set *S* that has the maximum value of *f(a)*.



# MINIMAX Algorithm

The functions used are:

- **GEN(Pos)** : This function generates a list of **SUCCs** (successors) of the Pos , where Pos represents a variable corresponding to position.
- **EVAL(Pos, Player)** : This function returns a number representing the **goodness of Pos for the player** from the current position.
- **DEPTH(Pos, Depth)** : It is a Boolean Function that **returns true** if the search has **reached the maximum depth** fro the current position , else it return false.



# MINIMAX Algorithm

```
MINIMAX(Pos, Depth, Player) {  
    • If DEPTH(Pos, Depth) then  
        return ( { Val=EVAL(Pos, Player), Path= Nil } )  
    Else {  
        • SUCC_List = GEN(Pos)  
        • If SUCC_List = Nil then  
            • return ( { Val=EVAL(Pos, Player), Path= Nil } )  
        Else {  
            • Best_Val = Minimum value returned by EVAL function  
            • For each SUCC ∈ SUCC_List Do {  
                • SUCC_Result = MINIMAX (SUCC, Depth + 1, ~Player )  
                • New_Value = - Val of SUCC_Result  
                • If NEW_Value > Best_Val then {  
                    • Best_Val = NEW_Value  
                    • Best_Path = Add(SUCC, Path of SUCC_Result)  
                }  
            }  
            • Return ({Val=Best_Val, Path = Best_Path})  
        }  
    }  
}
```



## Example: Tic-Tac-Toe Game

- If P is a win for MAX, then
  - ❖  $f(P) = n$ , (where n is a very large positive number)
- If P is a win for MIN, then
  - ❖  $f(P) = -n$ ,
- If P is not a winning position for either player, then
  - ❖  $f(P) = (\text{total number of rows, columns, and diagonals that are still open for MAX}) - (\text{total number of rows, columns, and diagonals that are still open for MIN})$

	O	
	X	



# Points on MINIMAX

- MINIMAX algorithm is a **depth first process**.
- The **partial solution** that clearly appears to be **worse** than the known solutions are discarded.
- Some **branches** of tree can be **ignored** without changing the final score of the root.



# Alpha – Beta Pruning

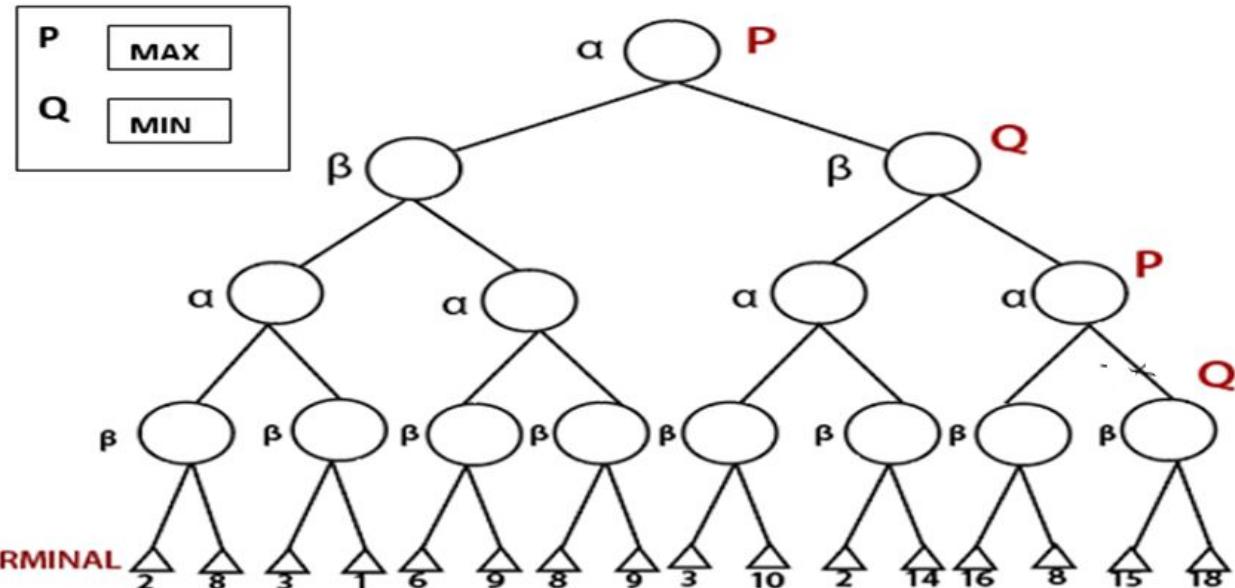


# Alpha – Beta Pruning

- In alpha-beta pruning it **cutoff the search** by exploring less number of nodes.
- It makes the same moves as a minimax algorithm does, but it prunes the unwanted branches using the **pruning technique**.
- Alpha-beta pruning works on two threshold values, i.e.,  $\alpha$  (**alpha**) and  $\beta$  (**beta**).
- $\alpha$  (**alpha**) : It is the best highest value, a MAX player can have. It is the lower bound, which represents negative infinity value.
- $\beta$  (**beta**) : It is the best lowest value, a MIN player can have. It is the upper bound which represents positive infinity.
- So, each MAX node has  $\alpha$  (**alpha**) -value, which never decreases, and each MIN node has  $\beta$  (**beta**) -value, which never increases.
- **The search is depth first and stops at any MIN node whose beta value is smaller than or equal to the alpha value of its parent.**
- **The search stops at any MAX node whose alpha value is greater than or equal to the beta value of its parent.**



# $\alpha - \beta$ Pruning



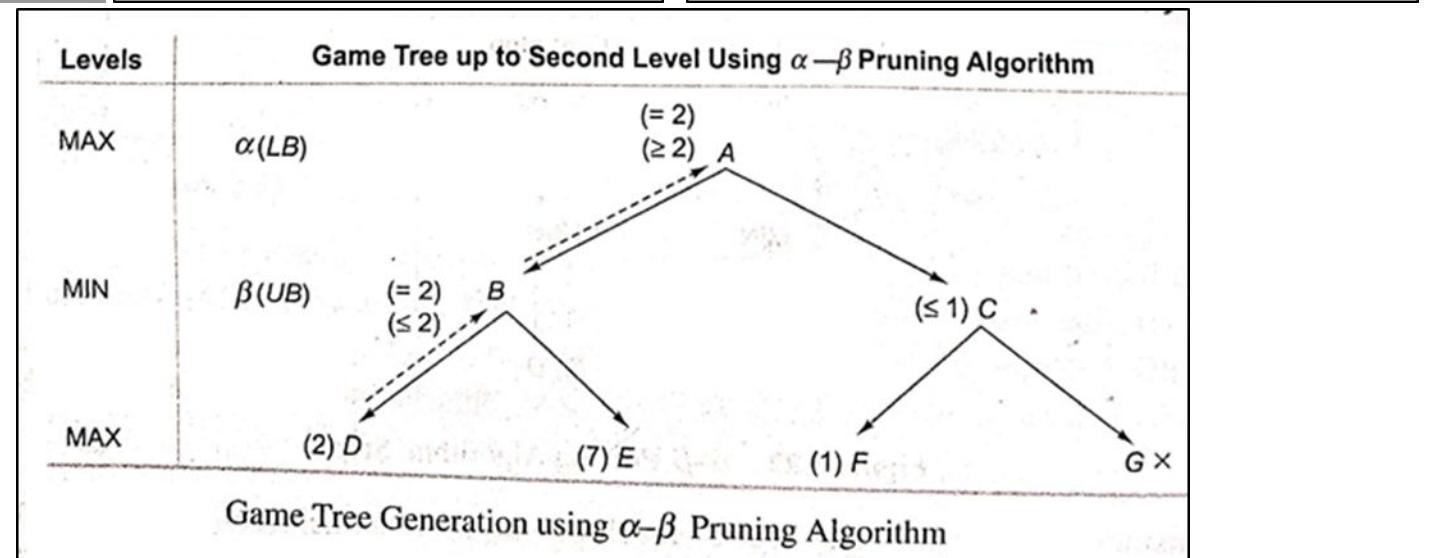
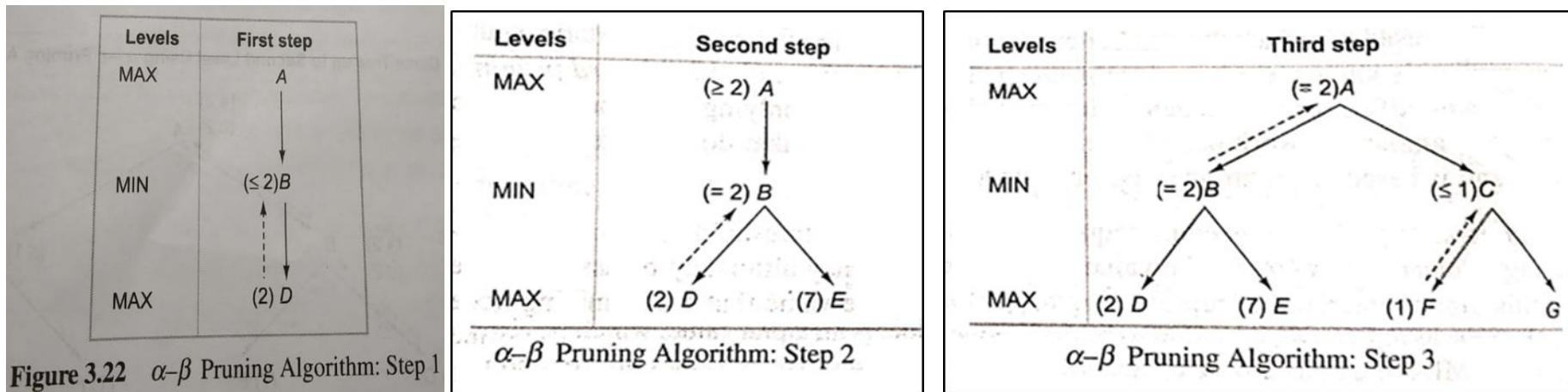
**Alpha-beta pruning**

- P and Q are two players.
- P be the player who will try to win the game by maximizing its winning chances.
- Q is the player who will try to minimize P's winning chances.
- $\alpha$  will represent the value for P(Maximum).
- $\beta$  will represent the value for Q(Minimum).



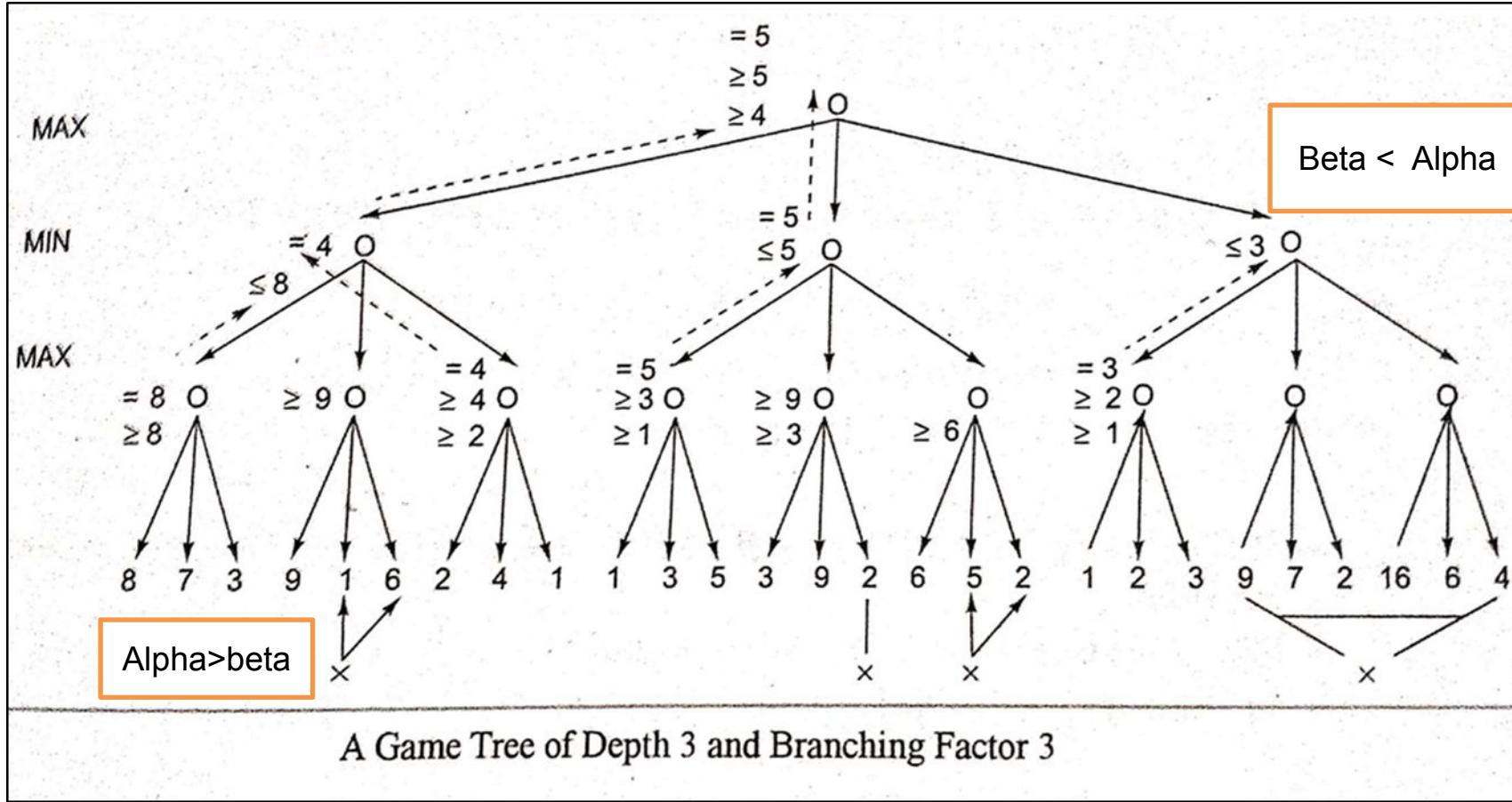
# Alpha – Beta Pruning

- Systematic Construction of game tree and propagation of  $\alpha$  and  $\beta$  values using Alpha – Beta Pruning.

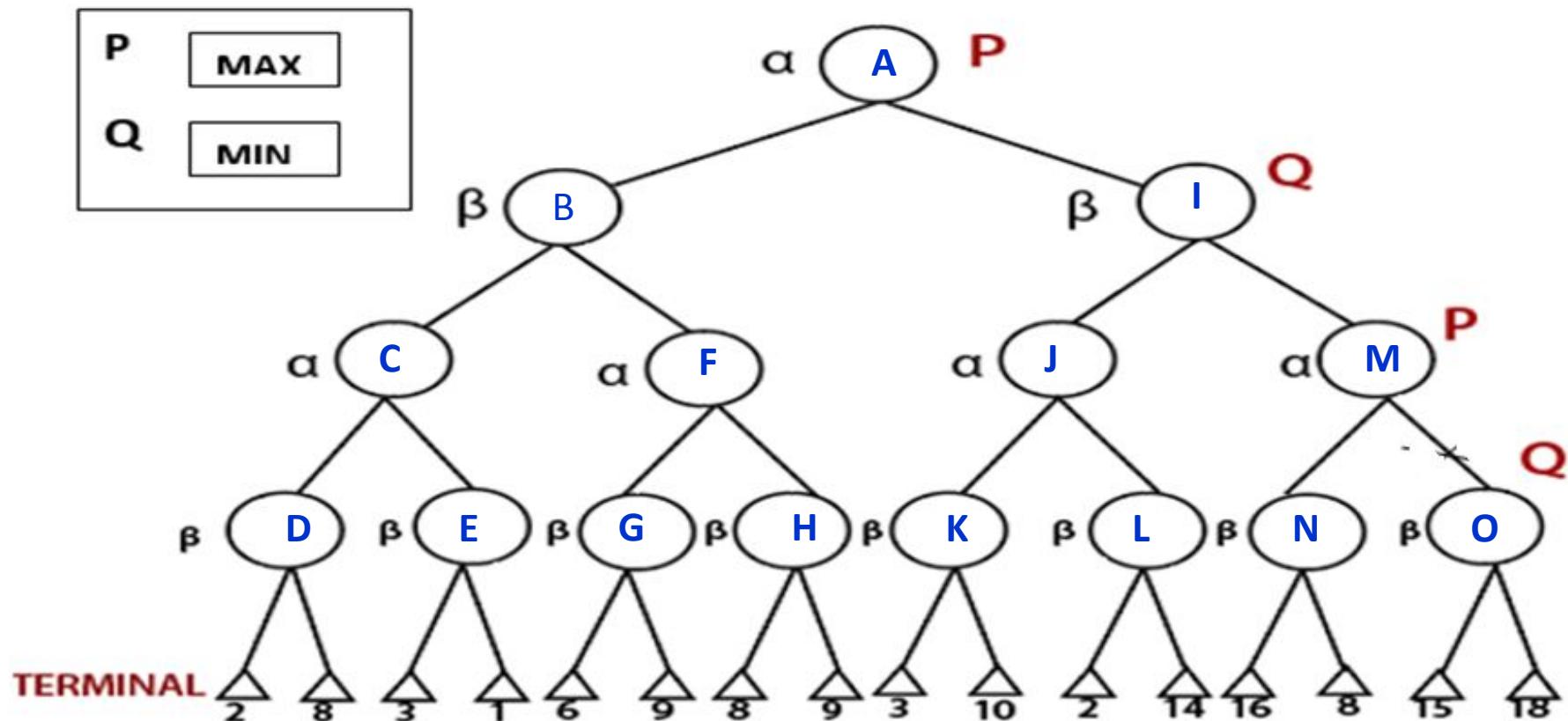




# Alpha – Beta Pruning



# Example



Alpha-beta pruning



# MINIMAX Algorithm using $\alpha - \beta$ Pruning

MINIMAX-  $\alpha\beta$ (Pos, Depth, Player, Alpha, Beta) {

- If DEPTH(Pos, Depth) then

return ( { Val=EVAL(Pos, Depth), Path= Nil } )

Else {

- SUCC\_List = GEN(Pos)

- If SUCC\_List = Nil then

• return ( { Val=EVAL(Pos, Depth), Path= Nil } )

Else {

- For each SUCC  $\in$  SUCC\_List Do {

• SUCC\_Result = MINIMAX -  $\alpha\beta$  (SUCC, Depth + 1,  $\neg$  Player,  $\neg$  Beta,  $\neg$  Alpha )

• New\_Value =  $\neg$  Val of SUCC\_Result

- If NEW\_Value > Beta then {

• Beta = NEW\_Value

• Best\_Path = Add(SUCC, Path of SUCC\_Result)

}

• if Beta  $\leq$  Alpha then return Do ( { Val=Beta, Path = Best\_Path } )

}

- Return ({Val=Best\_Val, Path = Best\_Path})

}

}

}

```
function ALPHA-BETA-SEARCH(state) returns an action
  v  $\leftarrow$  MAX-VALUE(state,  $-\infty$ ,  $+\infty$ )
  return the action in ACTIONS(state) with value v
```

---

```
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow -\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MAX(v, MIN-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\geq \beta$  then return v
     $\alpha \leftarrow \text{MAX}(\alpha, v)$ 
  return v
```

---

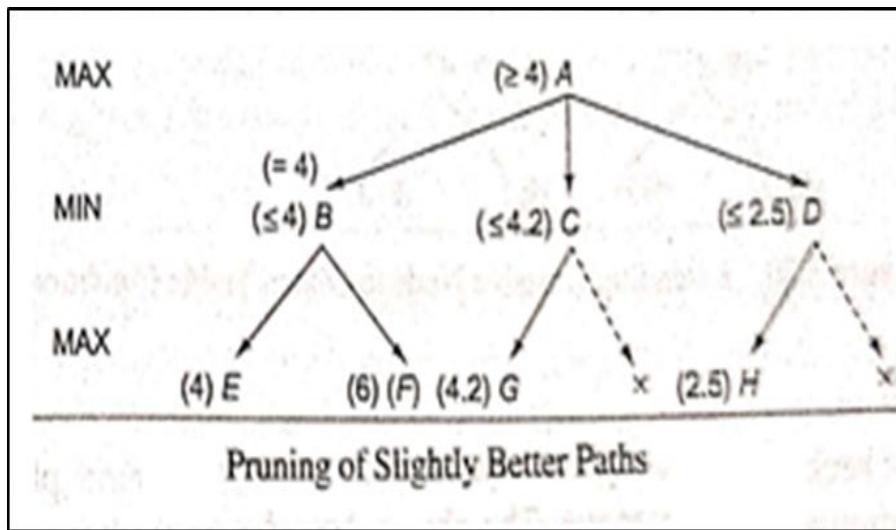
```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v  $\leftarrow +\infty$ 
  for each a in ACTIONS(state) do
    v  $\leftarrow$  MIN(v, MAX-VALUE(RESULT(s,a),  $\alpha$ ,  $\beta$ ))
    if v  $\leq \alpha$  then return v
     $\beta \leftarrow \text{MIN}(\beta, v)$ 
  return v
```

**Figure 5.7** The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain  $\alpha$  and  $\beta$  (and the bookkeeping to pass these parameters along).



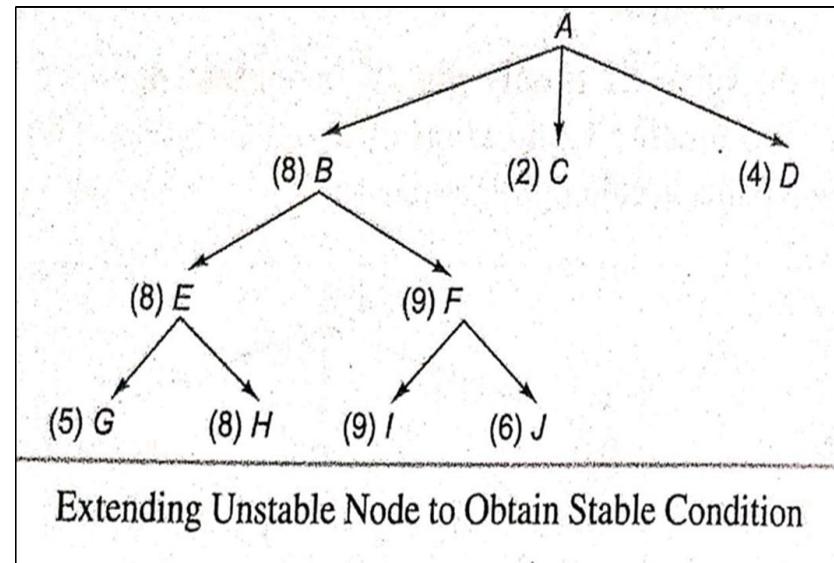
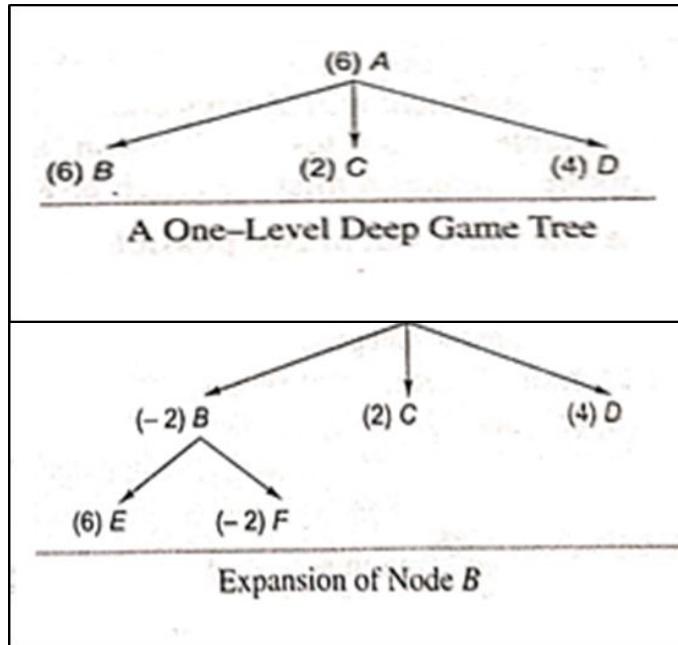
# Alpha Beta Pruning

- Refinements in  $\alpha$ - $\beta$  Pruning
  - Pruning of slightly better path: the value 4.2 is slightly better than 4, so terminate exploration of node C.
  - Terminating exploration of a sub tree that offers little possibility for improvement over known paths is called futility cut off.



- Iterative Deepening

- **Waiting for Quiescence:** Continue the search further until no drastic change occurs from one level to the next or till the condition is stable .
- Go deeper into the tree till the condition is stable and then make a decision regarding the move.
- Next level B again appears good.



- **Secondary search:** To provide double check, explore a game tree to an average depth of more ply and on the basis of that, choose a particular value.
- The chosen branch is then to be further expanded up to two levels to make sure that it still looks good.
- This technique is called Secondary search.
- **Alternative to  $\alpha$ - $\beta$  Pruning MINIMAX Procedure**

The MINIMAX procedure has some problem with all refinements.

- This is based on assumption that the opponent will always choose an optimal move.
- In winning situation it is fine but in a loosing situation, max may try other options and gain benefit if opponent makes a mistake.

**Iterative Deepening:** first try one ply...then two ply...and then 3 ply...

As per available time...



Thank  
You