Unit-4

Constraint Satisfaction Problems

What is Constraint Satisfaction Problems (CSP)?

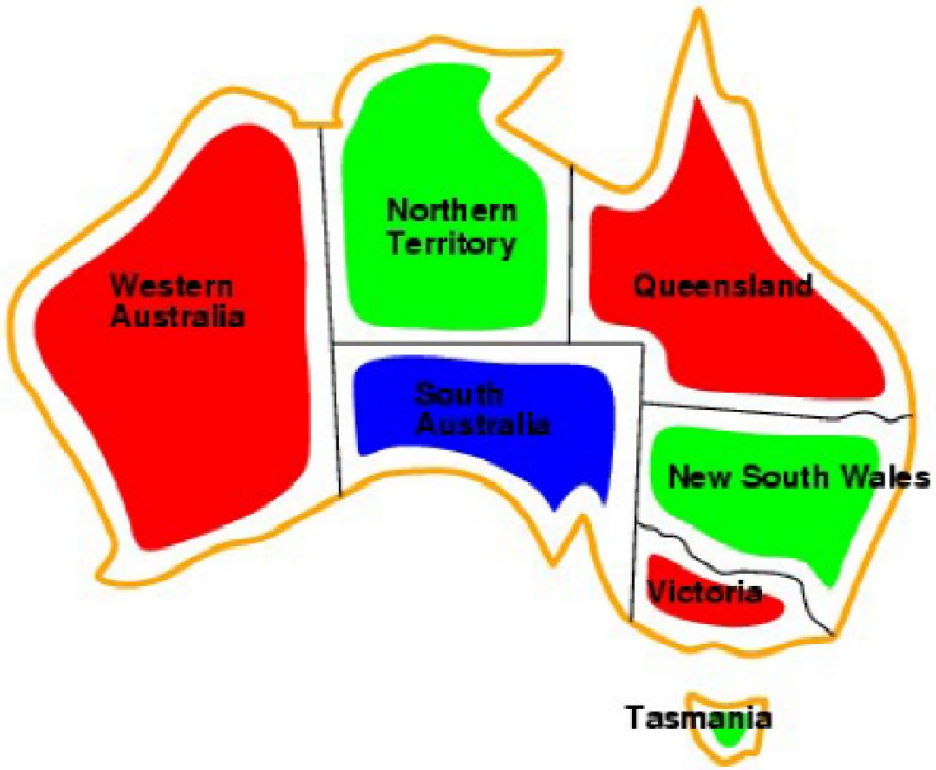
CSP is defined by 3 components (X, D, C):

- state: a set of variables X, each Xi , with values from domain Di
- goal test: a set of constraints C, each Ci involves some subset of the variables and specifies the allowable combinations of values for that subset
- Each constraint Ci consists of a pair <scope, rel>, where scope is a tuple of variables and rel is the relation, either represented explicitly or abstractly

- For example,
  - A and B are variables and respective domains are {1,2} and {2,4}
  - Constraint (A ≠ B)
  - The possible pairs are (1,2), (1,4), (2,4) but can not have (2,2)

Example: Map Coloring

- Variables: X = {WA, NT, Q, NSW, V, SA, T }
- Domains: Di = {red, green, blue}
- Constraints: adjacent regions must have different colors
- Solution?



{ WA = red, NT = green, Q = red, NSW = green,
V = red, SA = blue, T = red }

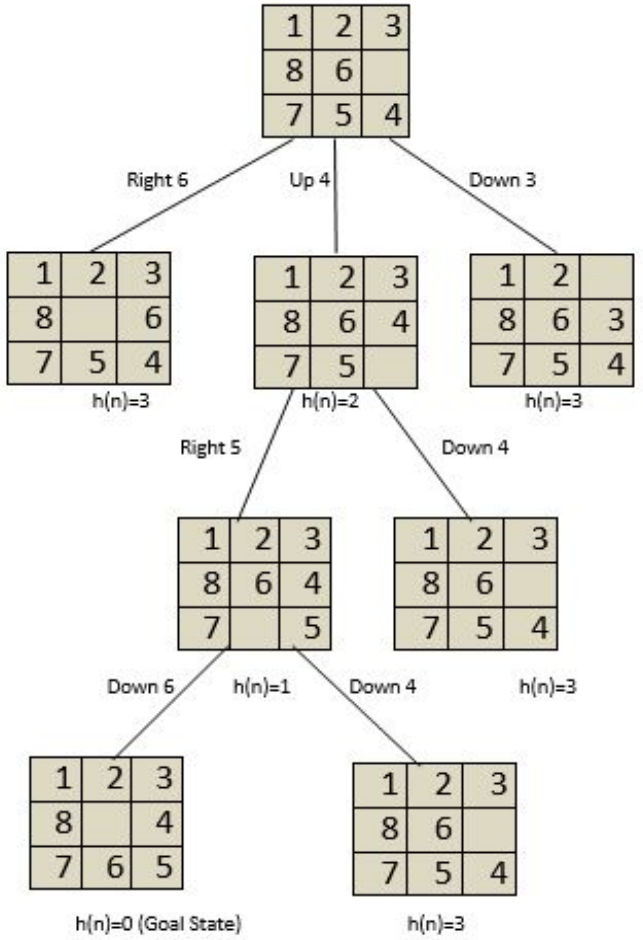
Local search (constraint satisfaction)

- In constraint satisfaction, local search is an **incomplete** method for finding a solution to a problem. It is based on iteratively **improving** an assignment of the variables until all constraints are satisfied.
- In particular, local search algorithms typically **modify** the value of a variable in an assignment at each step. The new assignment is close to the previous one in the space of assignment, hence the name **local search**.
- All local search algorithms use a **function** that evaluates the quality of assignment, for example the number of constraints violated by the assignment.
- This amount is called the cost of the assignment. The **aim of local search** is that of finding an assignment of minimal cost, which is a solution if any exists.

Local search (What is already covered?)

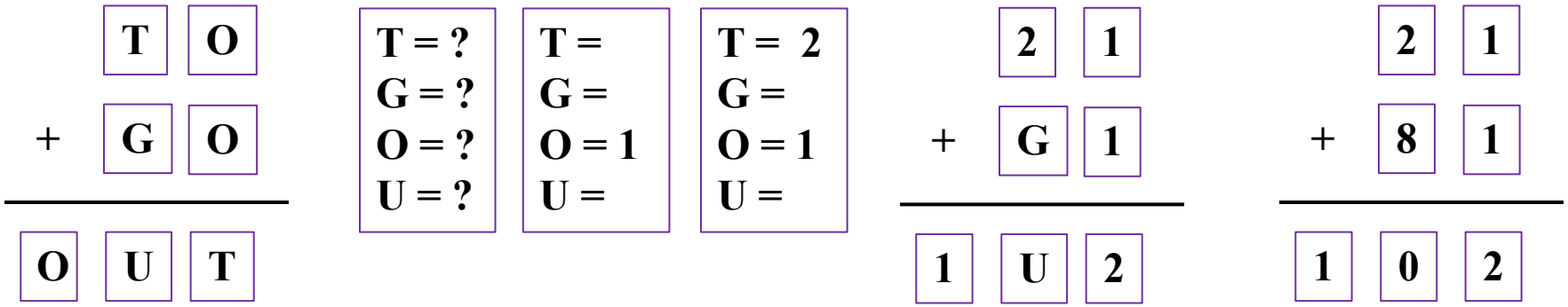
Two classes of **local search** algorithms exist.

- The **first** one is that of greedy or non-randomized algorithms. These algorithms proceed by changing the current assignment by always trying to decrease (or at least, non-increase) its cost.
- The main problem of these algorithms is the possible presence of plateaus **(see the figure)**, which are regions of the space of assignments where no local move decreases cost.
- The **second** class of local search algorithm have been invented to solve this problem. They escape these plateaus by doing random moves, and are called randomized local search algorithms. **(Simulated annealing)**



Crypt-arithmetic: Constraint Satisfaction Problem

- Crypt-arithmetic is a type of constraint satisfaction problem.
- Constraints:
  - No two letters have same value
  - Some of digit must be as shown in problem
  - There should be only one carry forward



Now G can be 8 or 9

If G is 9, then U must be 1, which is not possible as O is 1

So G must be 8 hence U is 0

T = 2
G = 8
O = 1
U = 0

	S	E	N	D
+	M	O	R	E
M	O	N	E	Y

M=1

C1=1	S	E	N	D
+	1	O	R	E
1	O	N	E	Y

If S=9
O=0

	9	E	N	D
+	1	0	R	E
1	0	N	E	Y

E + 0 = N, Which is not possible, hence (+1) + E + 0 = N (using carry)
Hence, 1 + E + 0= N

C2=1				
	9	E	N	D
+	1	0	R	E
1	0	N	E	Y

Now, N + R (+C3) = E,
that is N + R (+1) = E
Now N + R must be greater than 10 to generate a carry,
So N + R = E +10.
Now substitute the value of N from previous step

So E + 1 + R (+1) = E + 10
R = 10 - 1 = 9 if no carry
R = 10 - 2 = 8 if carry

Now 9 is already assigned so, <u>R must be equal to 8</u>

C3=1				
	9	E	N	D
+	1	0	8	E
1	0	N	E	Y

Now, D + E = Y
here carry is required so the possible values of D & E are:
5, 6, 7 (3, 4 can not be included because the sum gives 0 or 1 which is already assigned)
Let's assume D = 7 and E = 5
Then 7 + 5 = 12, which gives Y = 2

	9	5	6	7
+	1	0	8	5
1	0	6	5	2

Practice problems: Crypt-arithmetic

- A S + A = M O M
- S O + S O = T O O
- L E O + L E E = A L L
- B A S E + B A L L = G A M E S
- D O N A L D + G E R A L D = R O B E R T

Adversarial Search

- Adversarial search is search when there is an "enemy" or "opponent" changing the state of the problem every step **in a direction you do not want**.
- **Examples**: Chess, business, trading, war.
- You change state, but then you don't **control** the next state.
- Opponent will change the next state in a way:
  - ✔ **unpredictable**
  - ✔ **hostile** to you

Different Game Scenarios using Adversarial search

1. Games with Perfect information

These are open and transparent games where each player has all complete information on the status of the game, and the move of the opponent is quite visible to the players. Chess, Checkers, Ethello and Go are some of such games.

2. Games with Imperfect information

In these games, players will not have any information on the status of the game, and it will have blind moves by the players assuming certain conditions and predicting the outcomes. Tic-Tac-Toe, Battleship, Blind and Bridge are some examples of this type of game.

3. Deterministic Games

These games follow pre-defined rules and pattern, and it is played in a transparent manner. Each player can see the moves of the opponents clearly. Chess, Checkers, Tic-tac-toe, Go are some of the games played in this manner.

4. Non-Deterministic Games

Luck, Chance, Probability plays a major role in these types of game. The outcome of these games is highly unpredictable, and players will have to take random shots relying on luck. Poker, Monopoly, Backgammon are few of the games played in this method.

5. Zero-Sum games

These games are purely competitive in nature, and one player's gain is compensated by other player's loss. The net value of gain and loss is zero, and each player will have different conflicting strategies in these games. Depending on the game environment and game situation, each player will either try to maximize the gain or minimize the loss.

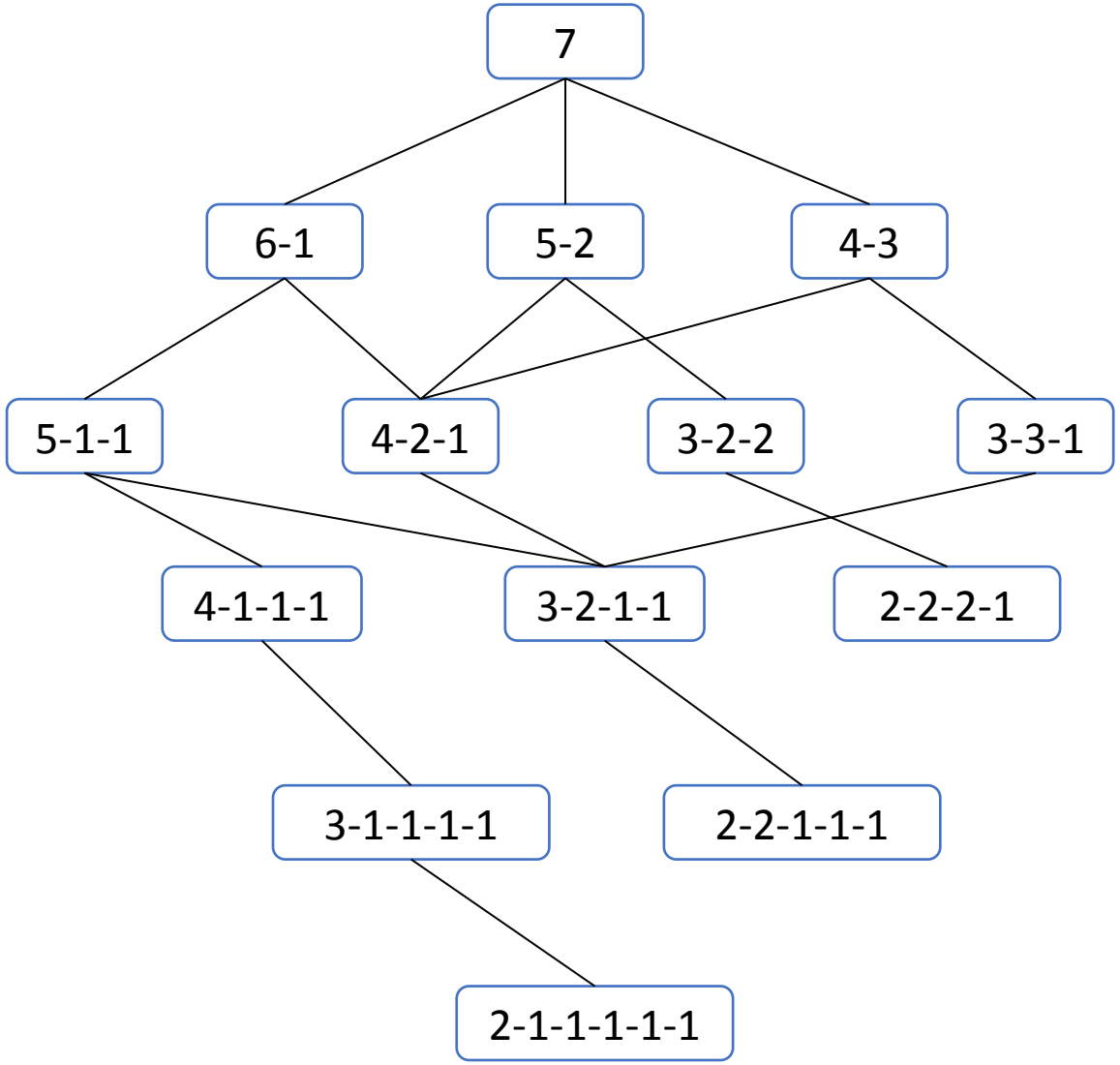
Games in AI

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **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.

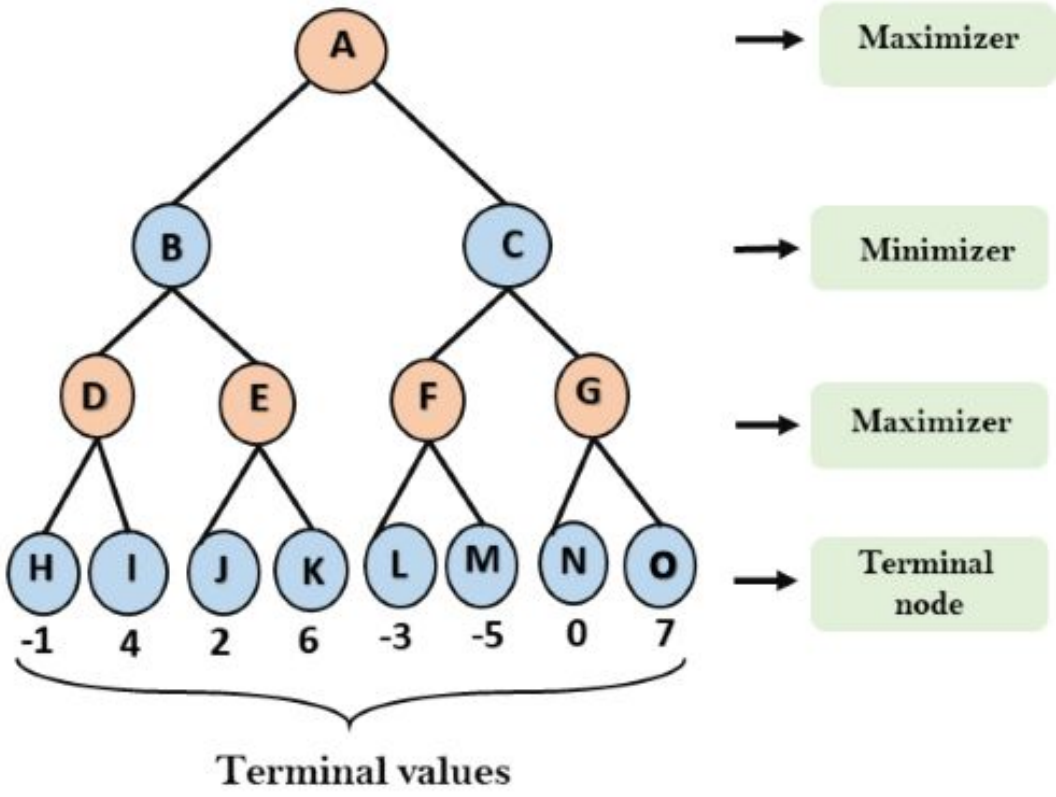
Minimax: Example of Adversarial Search

- The Minimax algorithm tries to predict the opponent's behaviour.
- It predicts the opponent will take the worst action from our viewpoint.
- We are MAX - trying to maximise our score/move to best state.
- Opponent is MIN - tries to minimise our score/move to worst state for us.
- **The Game of Nim:** At each move the player must divide a pile of tokens into 2 piles of different sizes. The first player who is unable to make a move loses the game.
- Example: 7 Piles



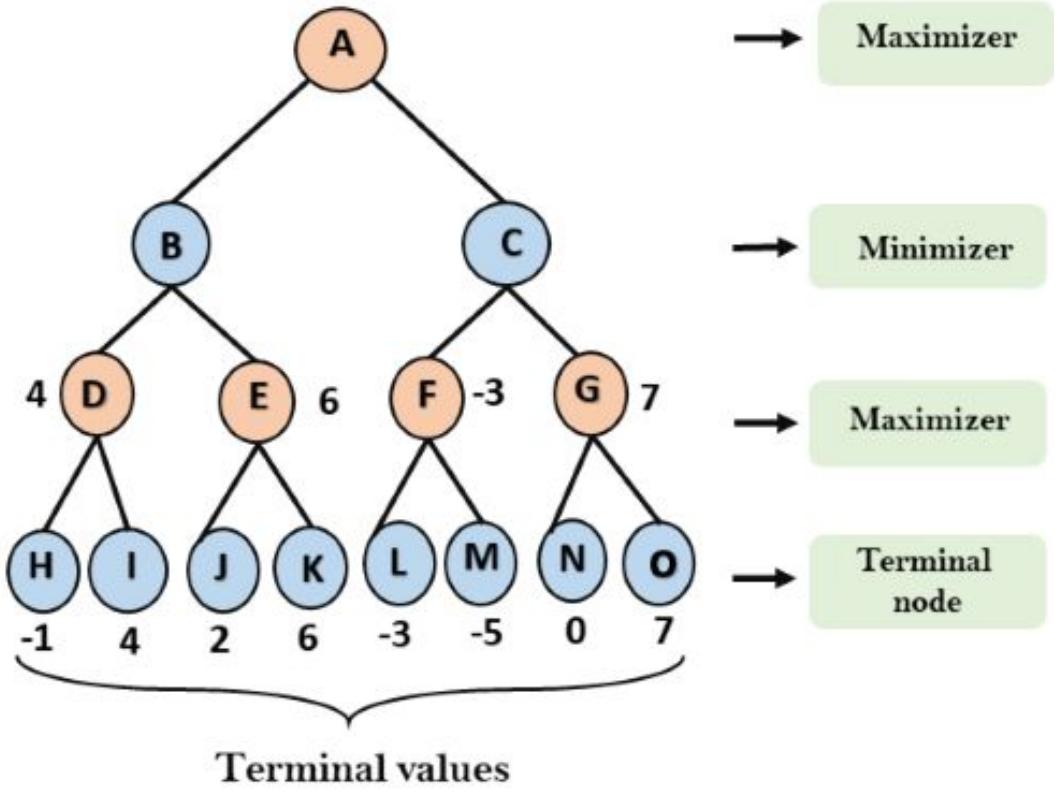
Minimax: another example

- **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 = -∞, and minimizer will take next turn which has worst-case initial value = +∞.



Cont'd

- **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

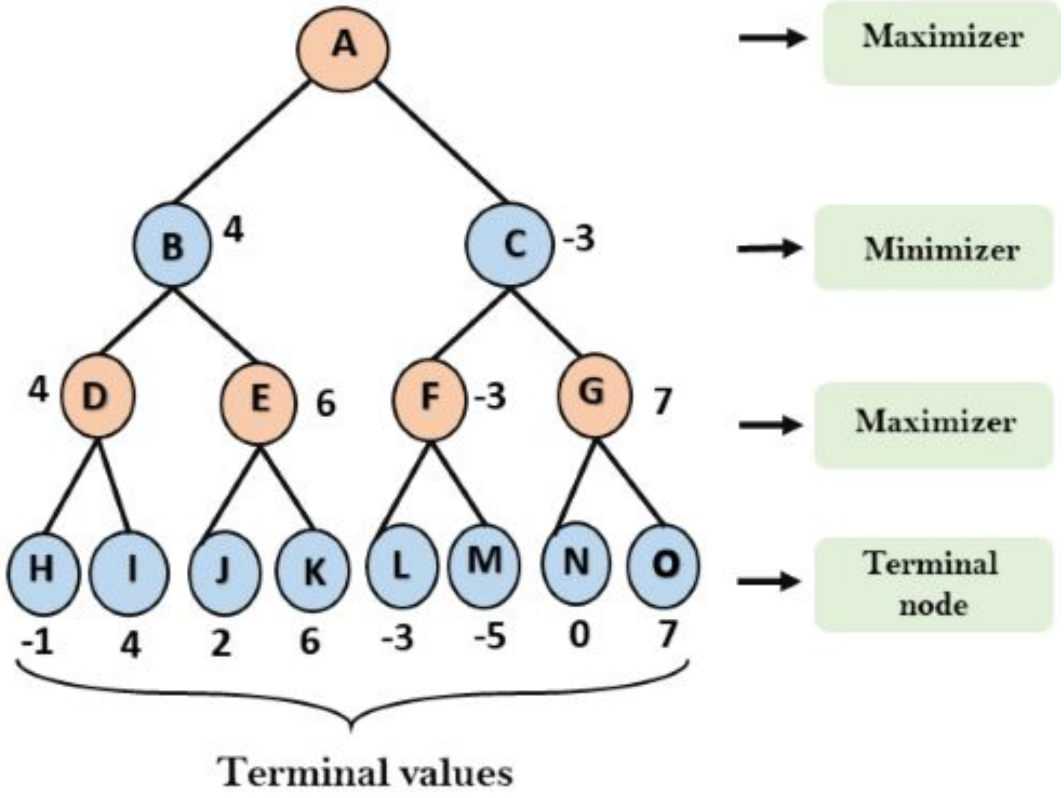
For Node F max(-3, -∞) => max(-3,-5) = -3

For node G max(0, -∞) => max(0, 7) = 7

Cont'd

- **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.

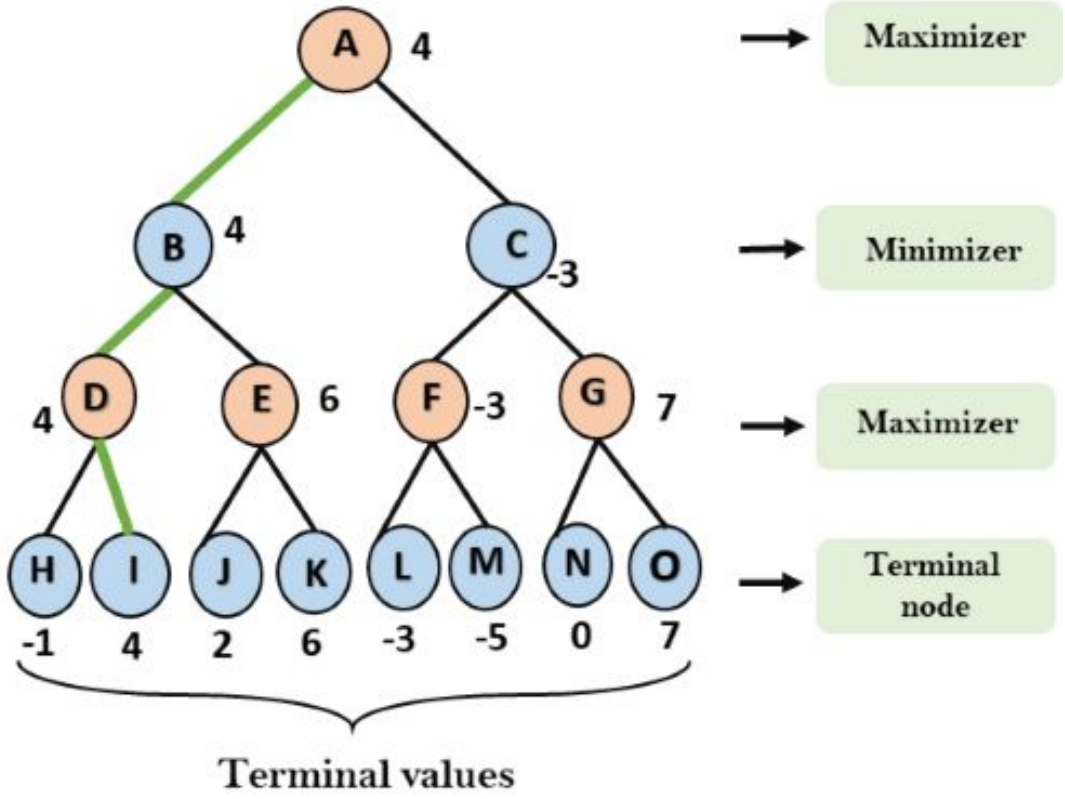
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



Cont'd

- **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



That was the complete workflow of the minimax two player game.

Steps for the minimax algorithm

In AI, Minimax algorithm can be stated as follows:

1. Create the entire game tree.
2. Evaluate the scores for the leaf nodes based on the evaluation function.
3. Backtrack from the leaf to the root nodes:
4. At the root node, choose the node with the maximum value and select the respective move.

For Maximizer, choose the node with the maximum score.

For Minimizer, choose the node with the minimum score.

Properties of minimax algorithm in AI

1. The algorithm is complete, meaning in a finite search tree, a solution will be certainly found.
2. It is optimal if both the players are playing optimally.
3. Due to Depth-First Search (DFS) for the game tree, the time complexity of the algorithm is $\mathbf{O(b^d)}$, where **b** is the branching factor and **d** is the maximum depth of the tree.
4. Like DFS, the space complexity of this algorithm is $\mathbf{O(b^d)}$**.**

Minimax: Advantages & Limitations

Advantages

A thorough assessment of the search space is performed.

Decision making in AI is easily possible.

New and smart machines are developed with this algorithm.

Limitations

Because of the huge branching factor, the process of reaching the goal is slower.

Evaluation and search of all possible nodes and branches degrades the performance and efficiency of the engine.

Both the players have too many choices to decide from.

If there is a restriction of time and space, it is not possible to explore the entire tree.

***** But with Alpha-Beta Pruning, the algorithm can be improved.***

```
function minimax(node, depth, maximizingPlayer)
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, false)
  maxEva= max(maxEva,eva) //Maximum of the values
return maxEva

else      // for Minimizer player
 minEva= +infinity
 for each child of node do
    eva= minimax(child, depth-1, true)
    minEva= min(minEva, eva)  // minimum of the values
 return minEva
```

Minimax using alpha-beta pruning

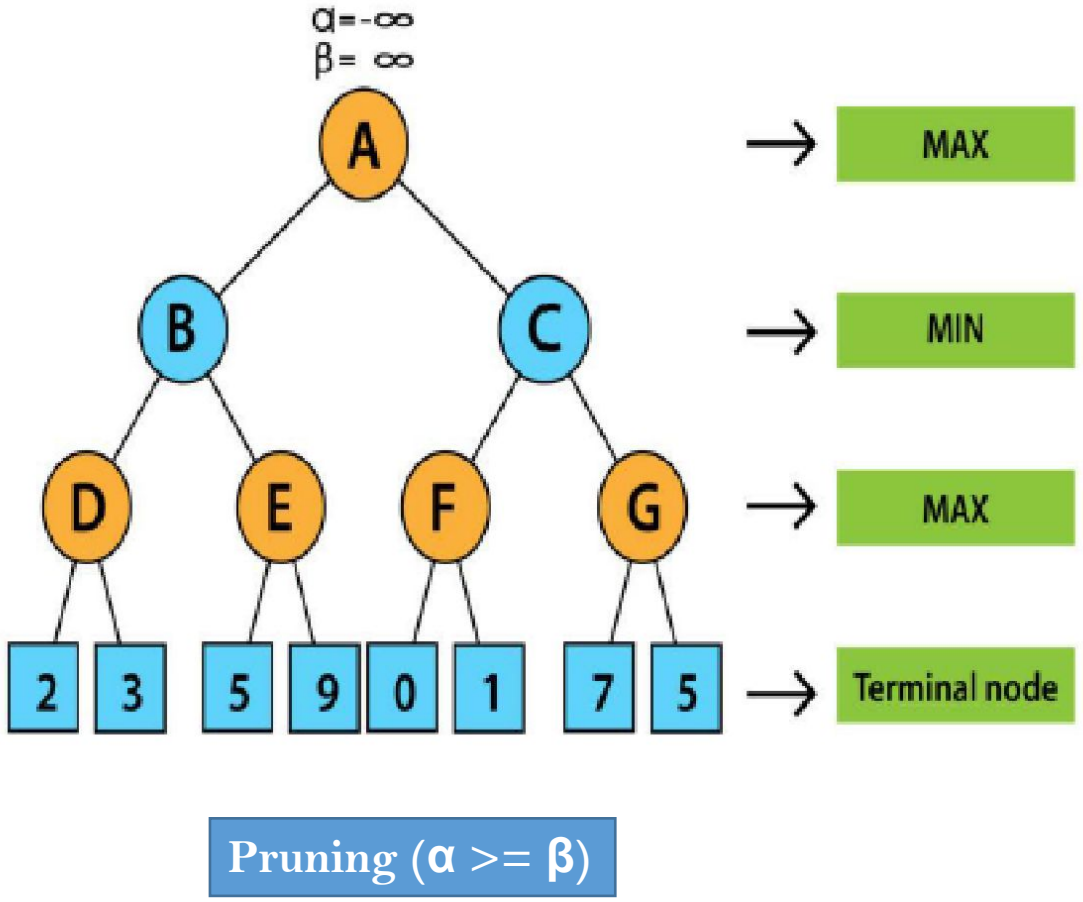
- Minimax procedure is a **depth-first** process
- One path is explored as far as time allows, the **static evolution function** is applied to the game positions at the last step of the path.
- The efficiency of the depth-first search can improve by **branch and bound** technique in which partial solutions that clearly worse than known solutions can abandon early.
- It is necessary to modify the branch and bound strategy to include **two bounds**, one for each of the players.
- This modified strategy called **alpha-beta pruning**.

Key points in Alpha-Beta Pruning

- **Alpha**: Alpha is the best choice or the **highest** value that we have found at any instance along the path of Maximizer. The initial value for alpha is $-\infty$.
- **Beta**: Beta is the best choice or the **lowest** value that we have found at any instance along the path of Minimizer. The initial value for alpha is $+\infty$.
- The condition for Alpha-beta **Pruning** is that **$\alpha >= \beta$**.
- Each node has to keep track of its alpha and beta values. **Alpha** can be updated only when it's **MAX's** turn and, similarly, **beta** can be updated only when it's **MIN's** chance.
- MAX will update only **alpha** values and MIN player will update only **beta** values.
- The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
- Alpha and Beta values only be passed to **child** nodes.

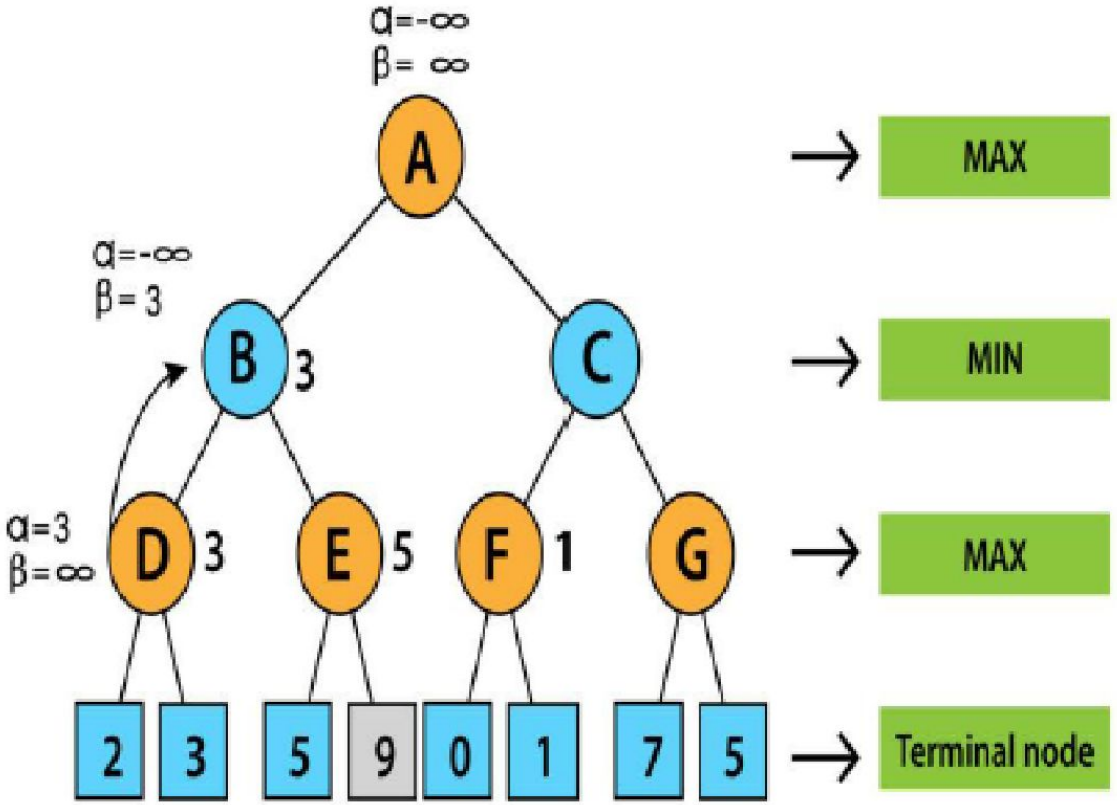
The working of Alpha-Beta Pruning - I

- We will first start with the initial move.
- We will initially define the **α** and **β** values as the **worst** case i.e. **α** = -∞ and **β**= +∞.
- We will **prune** the node only when **α** becomes greater than or equal to **β**. **(α >= β)**



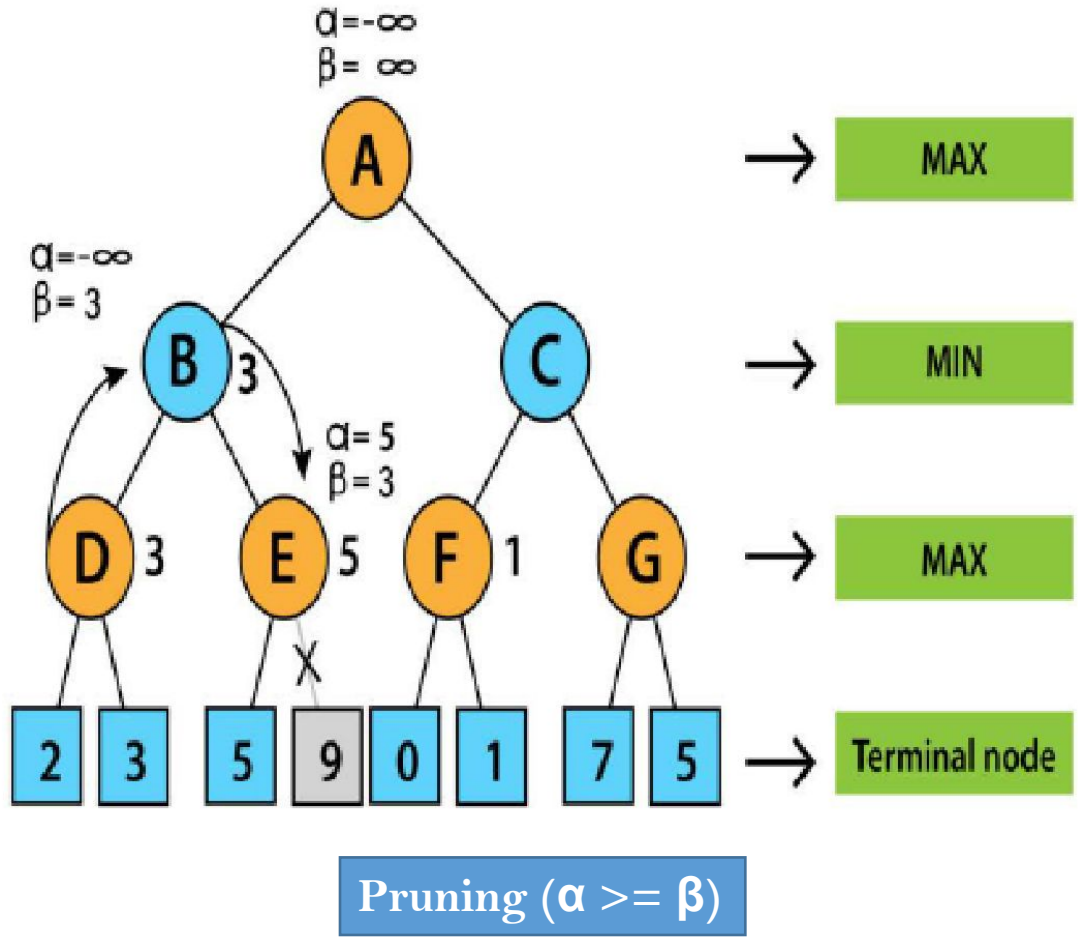
The working of Alpha-Beta Pruning - II

- Since the initial value of **α** is less than **β** so we **didn't prune** it. Now it's turn for MAX. So, at node D, value of **α** will be calculated. The value of **α** at node D will be max (2, 3). So, value of **α** at node **D will be 3**.
- Now the next move will be on node B and its turn for MIN now. So, at node B, the value of **α, β** will be min (3, ∞). So, at node B values will be **alpha= –** ∞ and **β = 3**.
- In the next step, algorithms traverse the next successor of Node B which is node E, and the values of **α**= -∞, and **β** = 3 will also be passed.



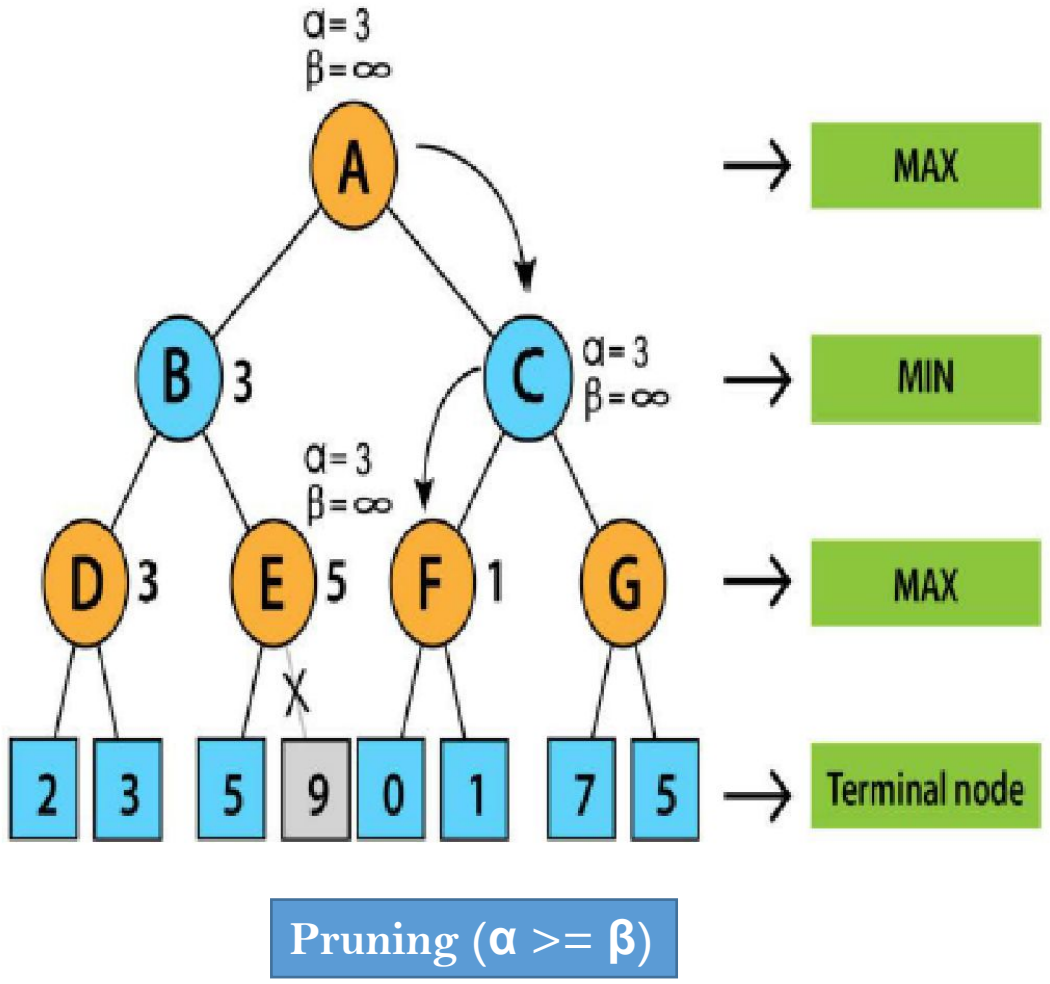
The working of Alpha-Beta Pruning - III

- Now it's turn for MAX. So, at node E we will look for MAX. The current value of **α** at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5.
- So, at node E, **α** = 5, **β** = 3.
- Now as we can see that **α** is greater than **β** which is satisfying the pruning condition.
- So we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.



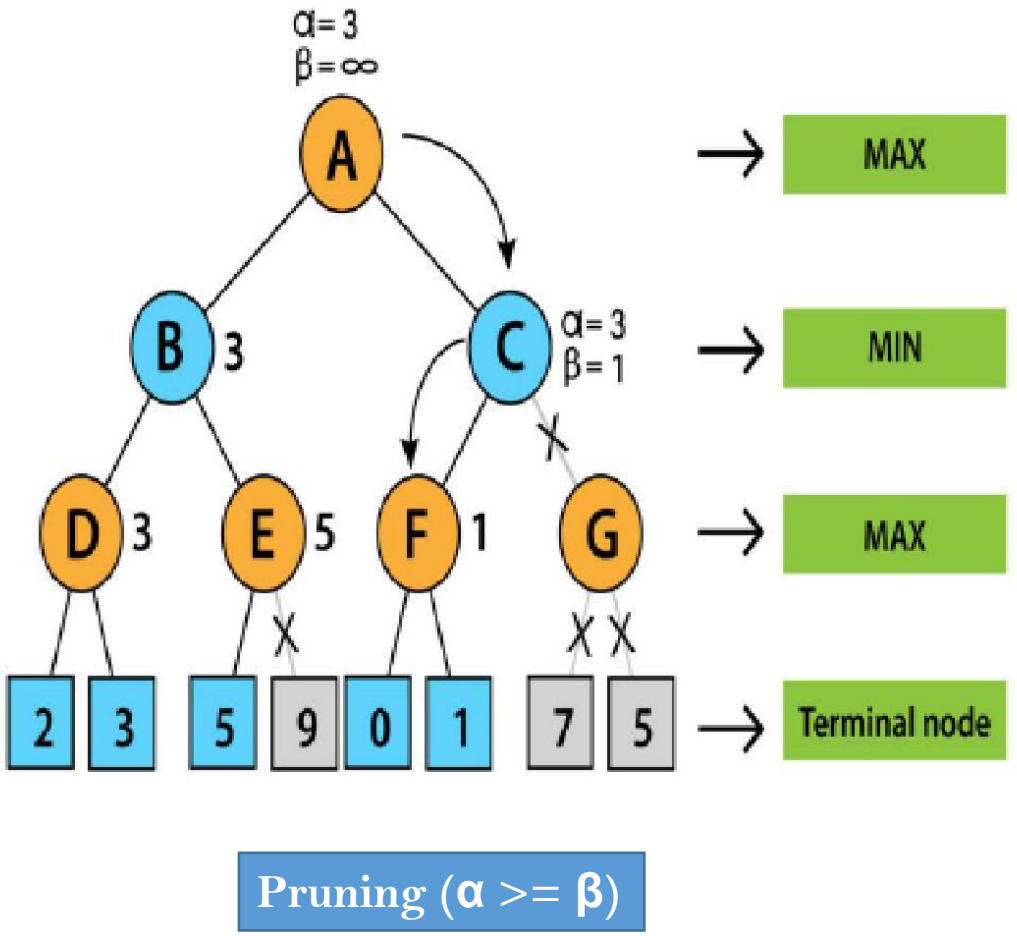
The working of Alpha-Beta Pruning - IV

- In the next step the algorithm again comes to node A from node B. At node A, **α** will be changed to maximum value as MAX (- ∞, 3).
- So now the value of **α** and **β** at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.
- At node F the value of **α** will be compared to the left branch which is 0.
- So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still **α** remains 3, but the node value of F will become 1.



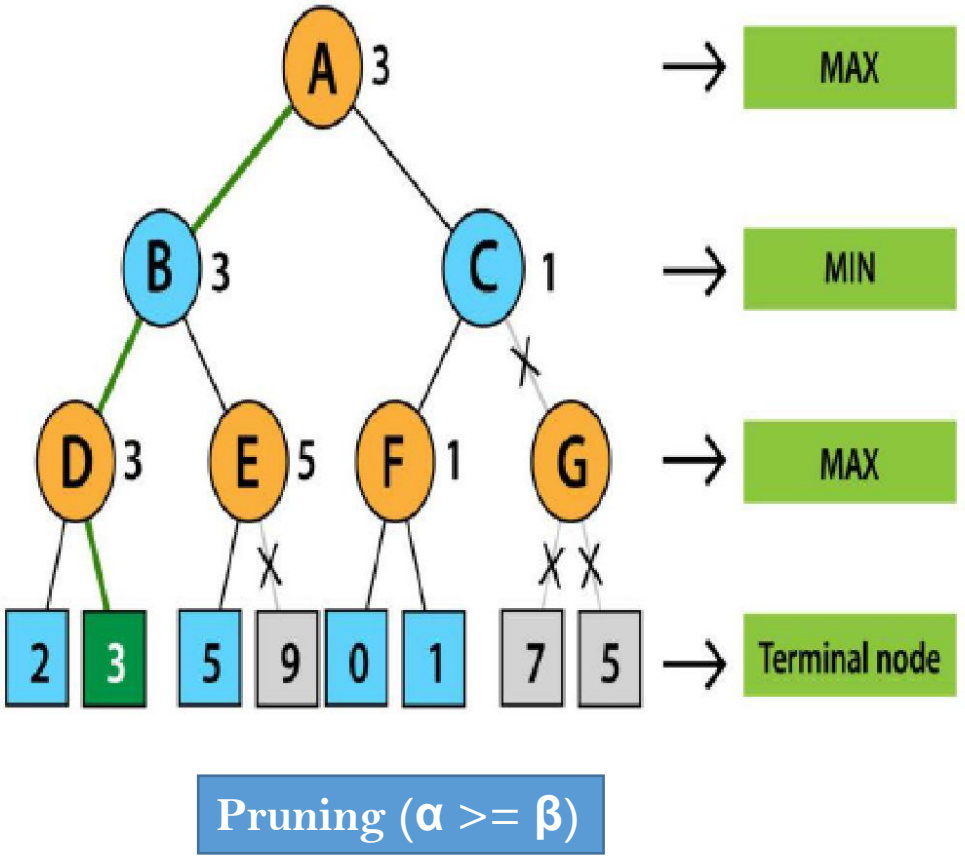
The working of Alpha-Beta Pruning - V

- Now node F will return the node value 1 to C and will compare to **β** value at C. Now its turn for MIN.
- So, MIN (+ ∞, 1) will be 1. Now at node C, **α**= 3, and **β**= 1 and **α** is greater than **β** which again satisfies the pruning condition.
- So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.



The working of Alpha-Beta Pruning - VI

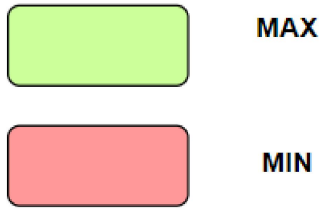
- Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.
- The represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.

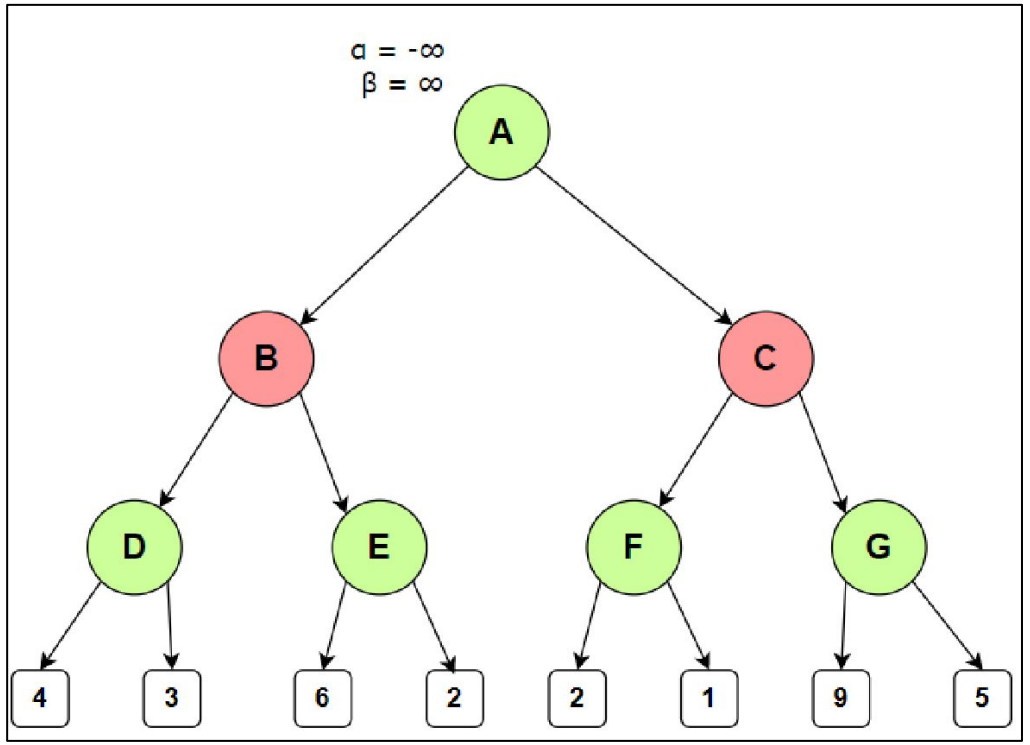


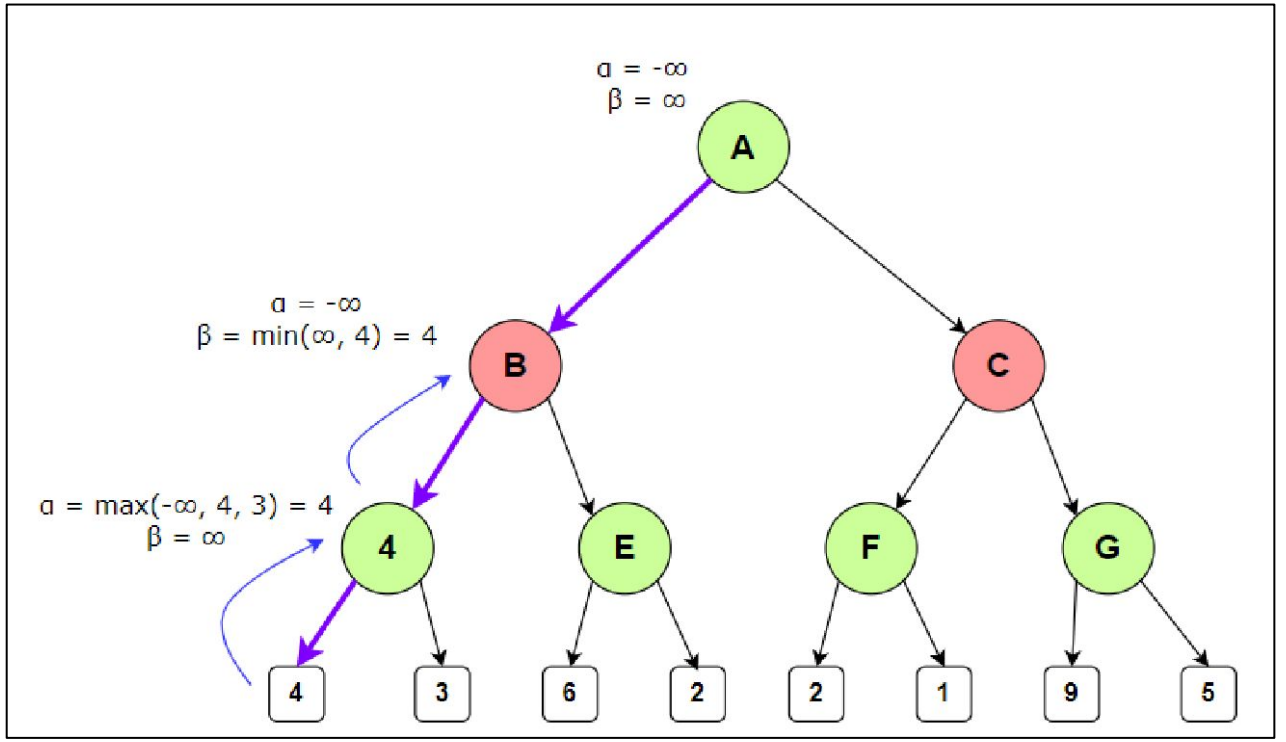
function **minimax**(node, depth, isMaximizingPlayer, alpha, beta):

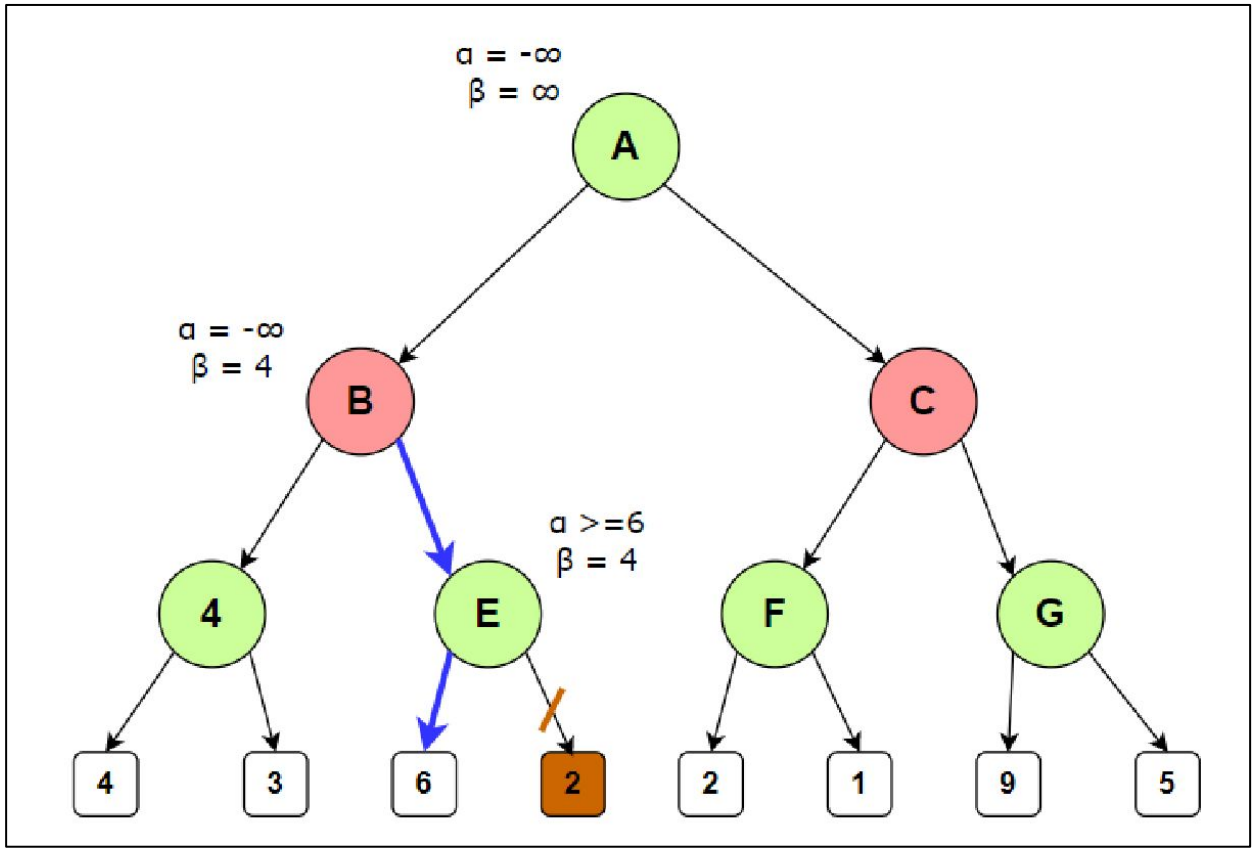
```
if node is a leaf node :
    return value of the node
if isMaximizingPlayer :
    bestVal = -INFINITY
    for each child node :
        value = minimax(node, depth+1, false, alpha, beta)
        bestVal = max( bestVal, value)
        alpha = max( alpha, bestVal)
        if beta <= alpha:
            break
    return bestVal
```

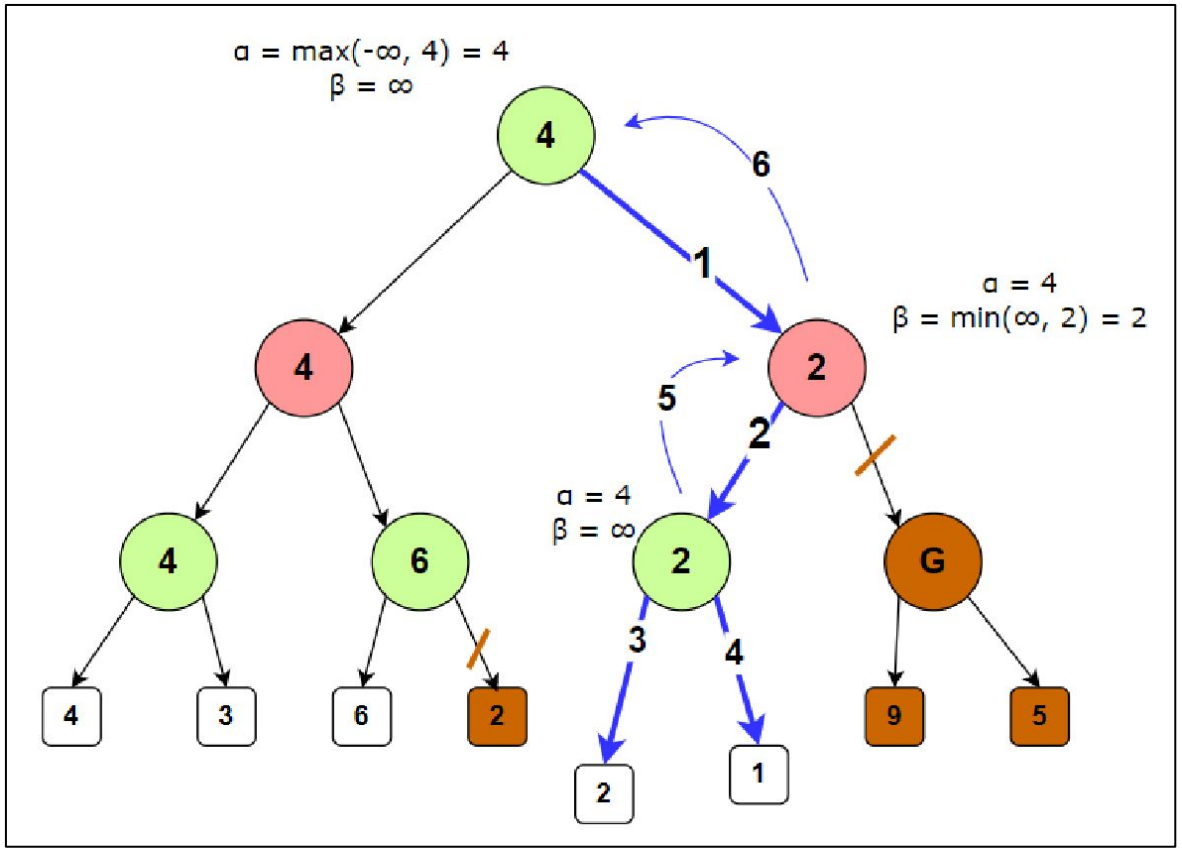
Example: Alpha-Beta Pruning



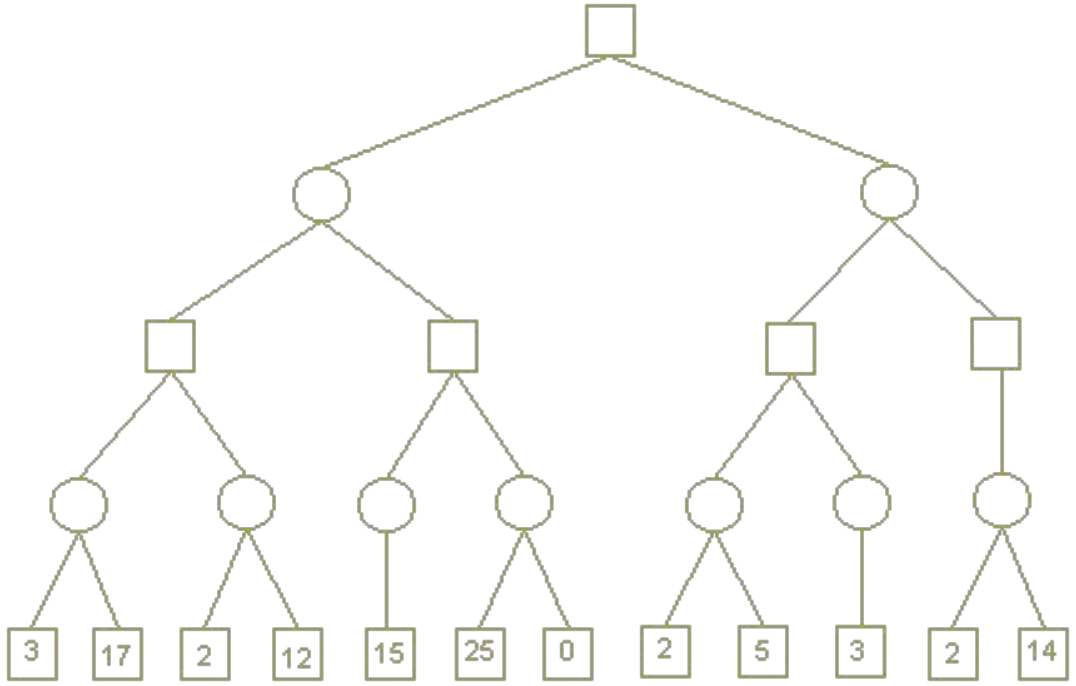


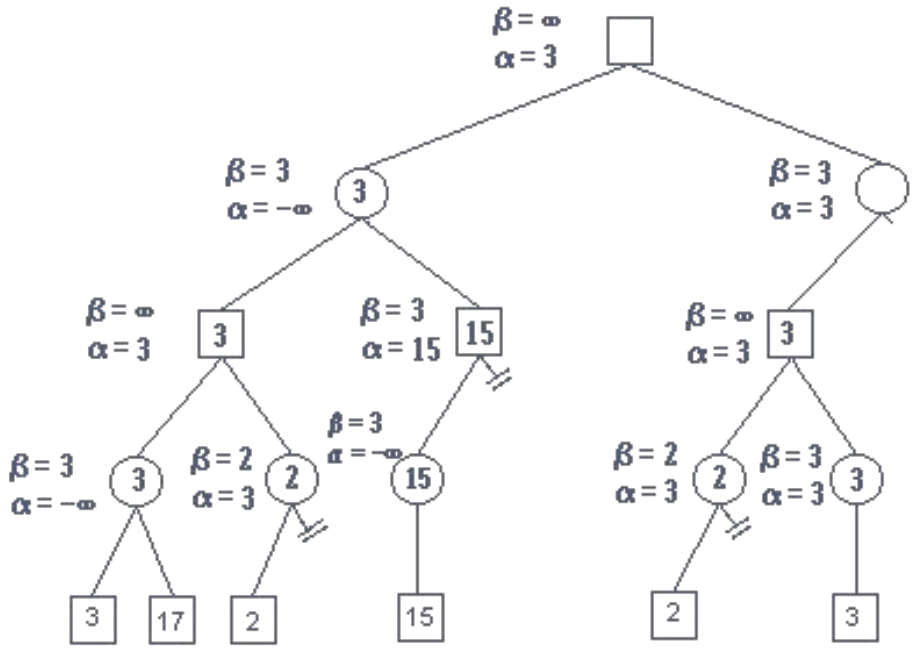



MAX
MIN
α = -∞
β = ∞
A
α = -∞
β = 4
B
C
α >=6
β = 4
4
E
F
G
4
3
6
2
2
1
9
5

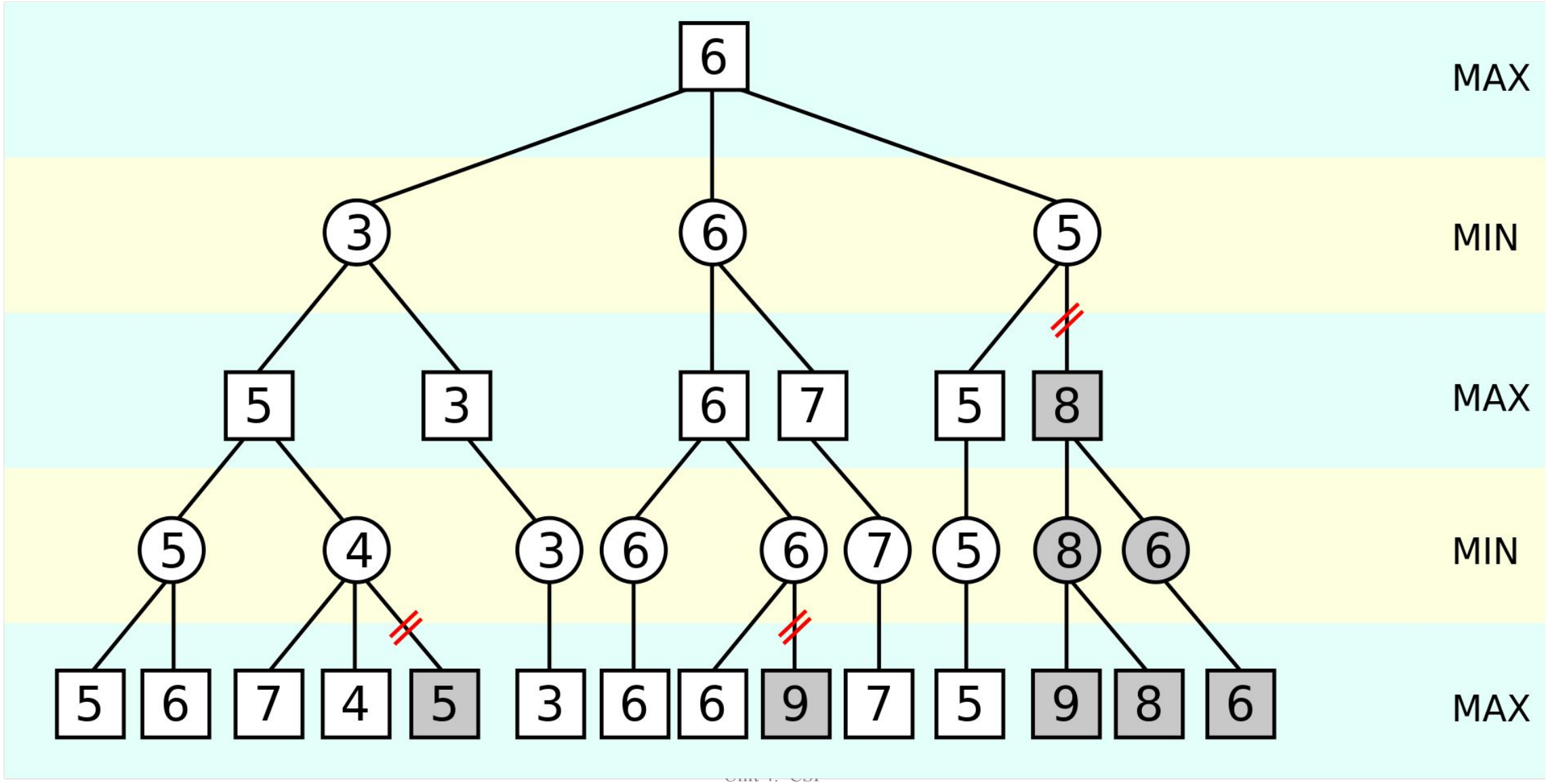

α = max(-∞, 4) = 4
β = ∞
4
6
1
α = 4
β = min(∞, 2) = 2
4
2
5
2
α = 4
β = ∞
4
6
2
G
3
4
4
3
6
2
2
1
9
5

Another Example: Alpha-Beta Pruning





Another Example: Alpha-Beta Pruning



Additional Refinements (for MiniMax)

- We have seen how game playing domain is different than other domains and how one needs to change the method of search. We have also seen how MiniMax search algorithm is applied and also seen how alpha beta pruning helps improve the efficiency of the MiniMax search algorithm.
- In fact it is possible to improve the functioning of algorithm by providing few **additional refinements** to the process of searching.
- There are a few simple **refinements** one can deploy while playing game playing algorithm.
  - **First** is waiting for stable states where the change of heuristic values across layers is not drastic.
  - **Second** is to use secondary search to avoid horizon effect.
  - **Third** is to use book moves for typical non-search part of the game playing.
  - **Fourth** is to use other than MiniMax algorithm.

Assignment

- Explain and implement Tic-Tac-Toe game using MiniMax Algorithm