Game Playing

Game Playing
Game Playing
Game Playing

Course Name: Artificial Intelligence

Course code: CSE-403 [SECTION - A]

Outline

- Introduction To Game Playing
- Mini-max Algorithm
  - Working Procedure Of Mini-max Algorithm
  - **Examples Problems Mini-max Algorithm**
  - Properties Of Mini-max Algorithm
  - Limitations Mini-max Algorithm
- Alpha-beta Pruning Algorithm
  - Working Of Alpha-beta Pruning
  - **Example Problems Of Alpha-beta Pruning**
  - Move Ordering Of Alpha-beta Pruning

Game Playing in Artificial Intelligence

Game Playing is an important domain of artificial intelligence. Games don't require much knowledge; the only knowledge we need to provide is the rules, legal moves and the conditions of winning or losing the game. Both players try to win the game. So, both of them try to make the best move possible at each turn. Searching techniques like BFS(Breadth First Search) are not accurate for this as the branching factor is very high, so searching will take a lot of time.

So, we need another search procedures that improve –

- **Generate procedure** so that only good moves are generated.
- **Test procedure** so that the best move can be explored first.

Game playing is a popular application of artificial intelligence that involves the development of computer programs to play games, such as chess, checkers, or Go. The goal of game playing in artificial intelligence is to develop algorithms that can learn how to play games and make decisions that will lead to winning outcomes.

One of the earliest examples of successful game playing AI is the chess program Deep Blue, developed by IBM, which defeated the world champion Garry Kasparov in 1997. Since then, AI has been applied to a wide range of games, including two-player games, multiplayer games, and video games.

There are two main approaches to game playing in AI, rule-based systems and machine learning-based systems.

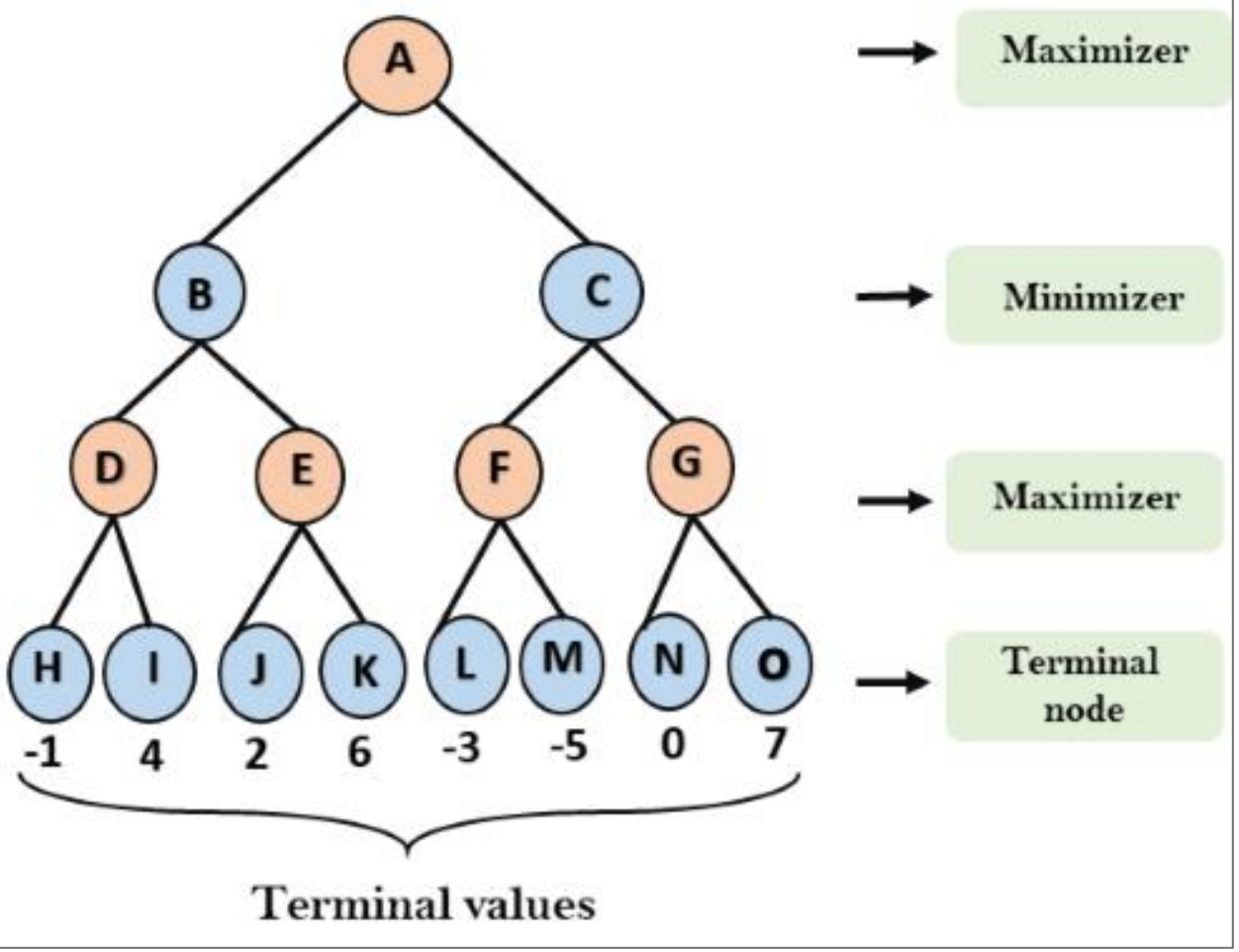
1. **Rule-based systems** use a set of fixed rules to play the game.
2. **Machine learning-based systems** use algorithms to learn from experience and make decisions based on that experience.

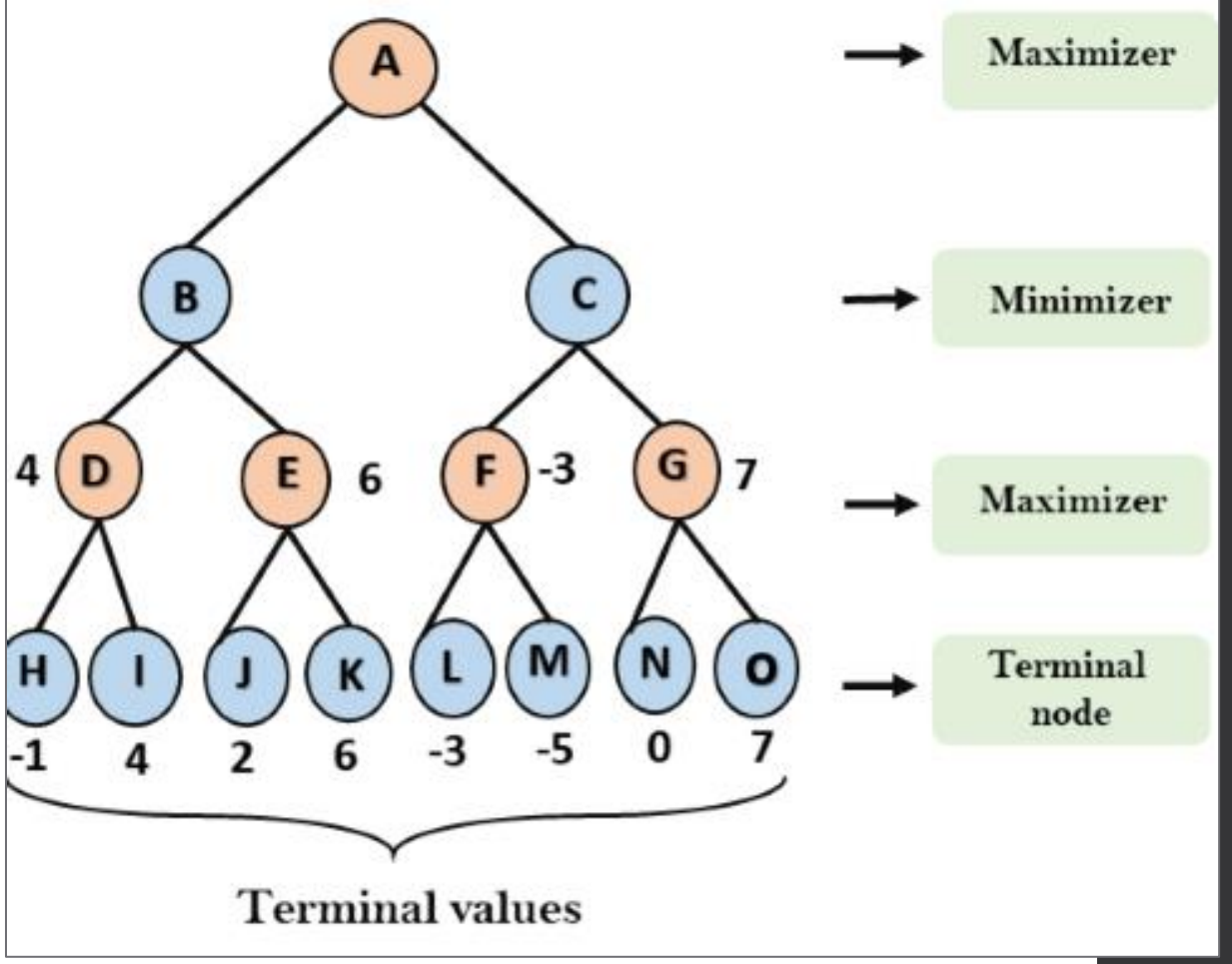
Mini-Max Algorithm in Artificial Intelligence

- Mini-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.
- Mini-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 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. 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 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.

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game. In this example, there are two players one is called Maximizer 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.
- **Example-1:** 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 below 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.





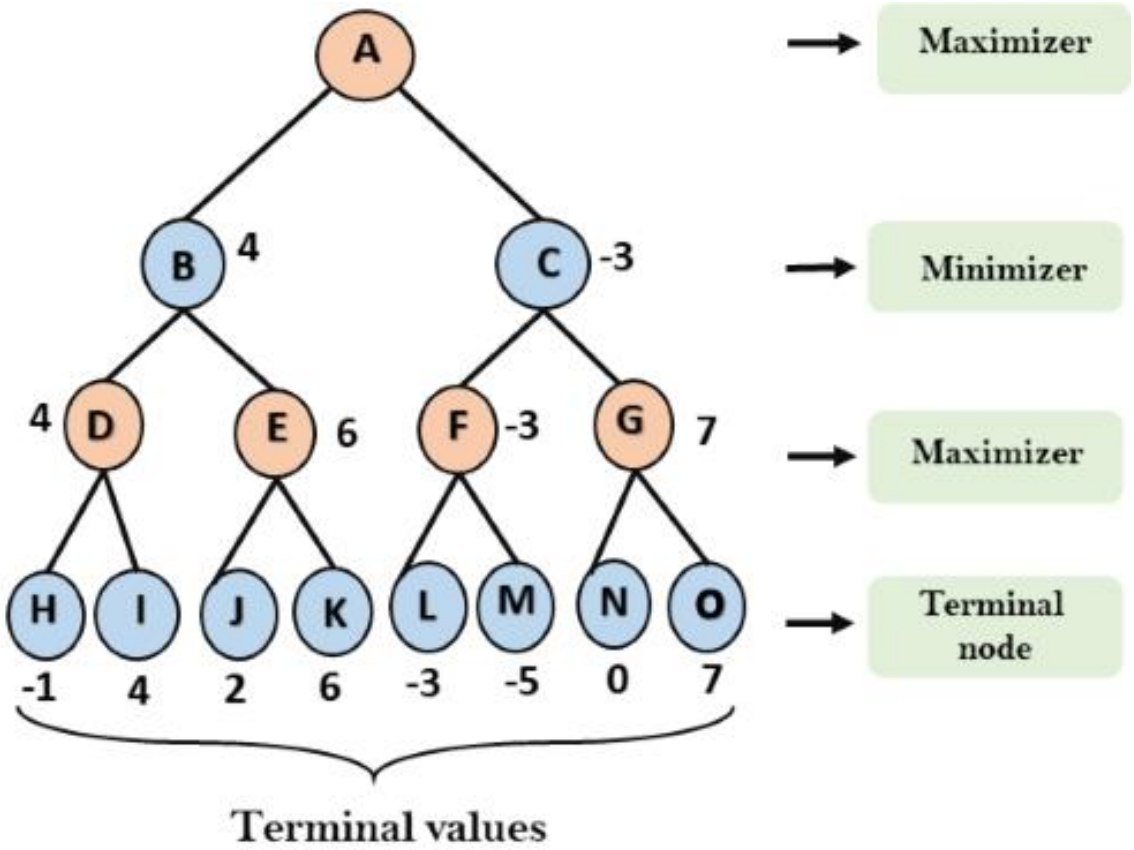
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

- 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.

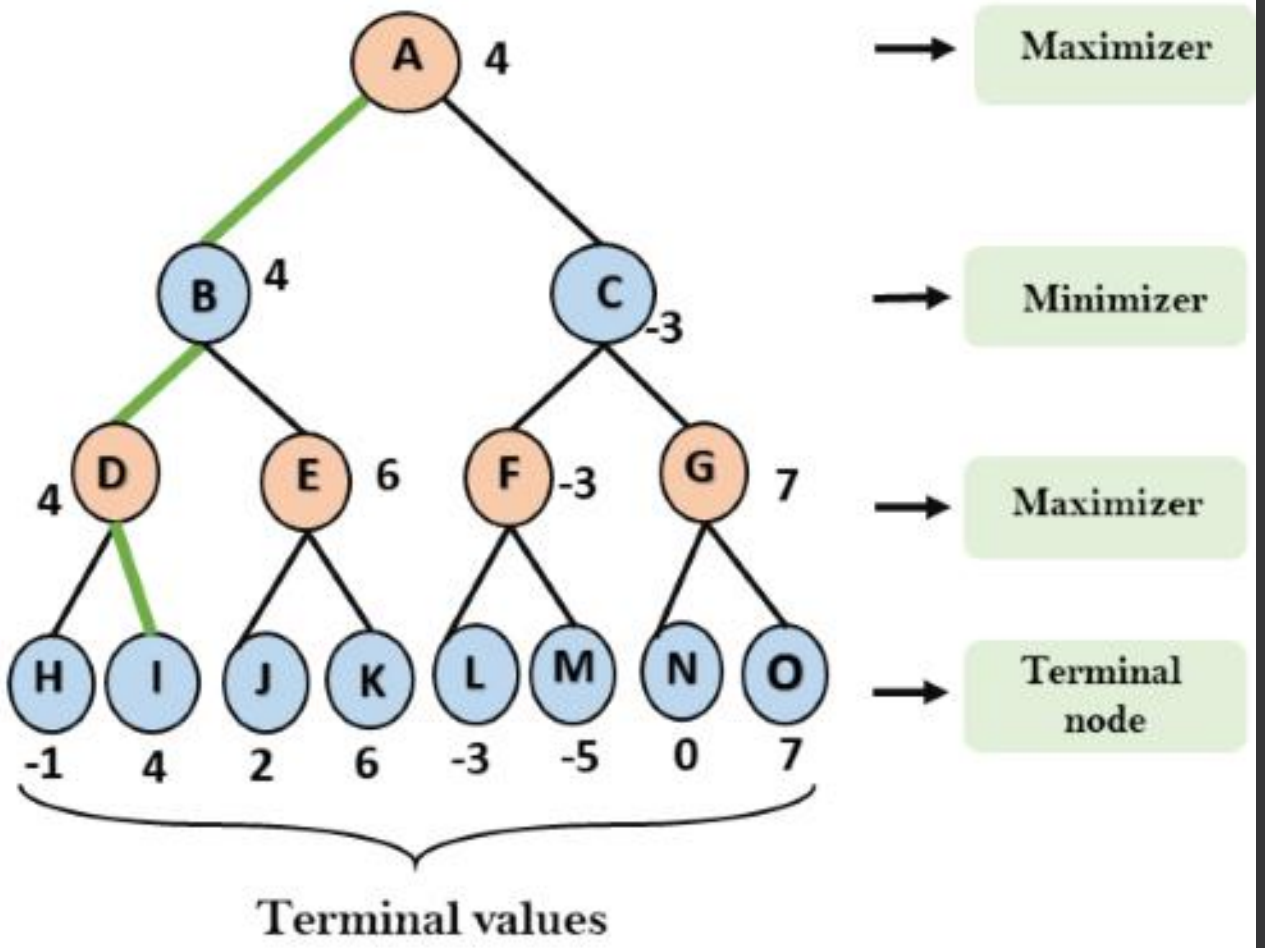
•For node B= min(4,6) = 4

•For node C= min (-3, 7) = -3

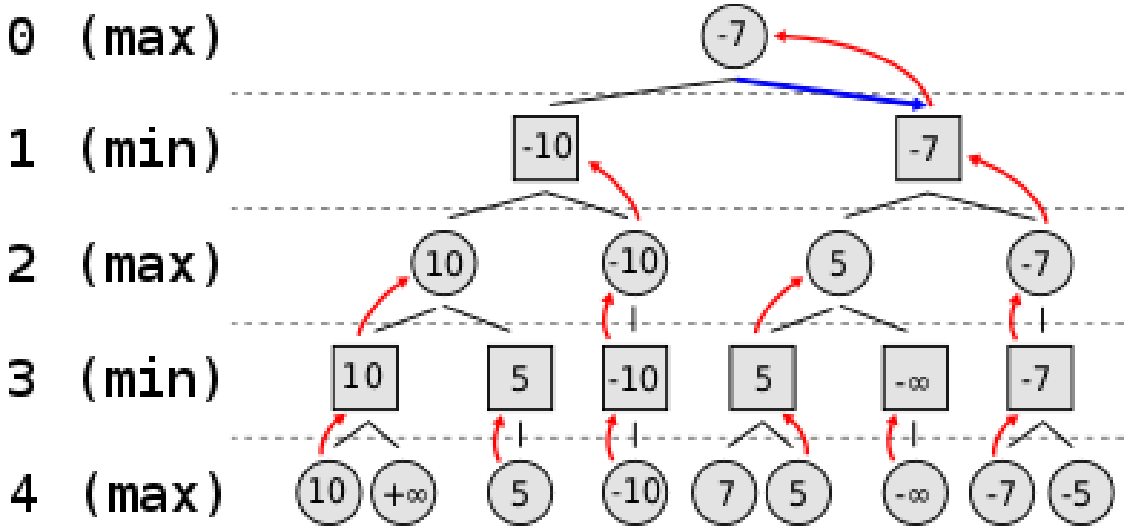


Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

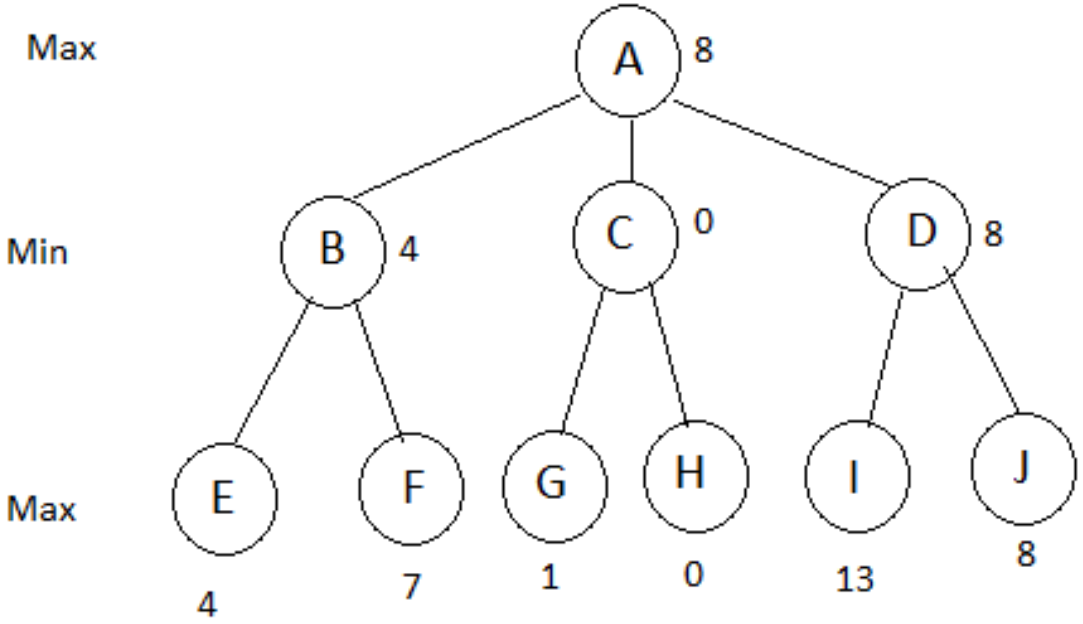
•For node A max(4, -3)= 4



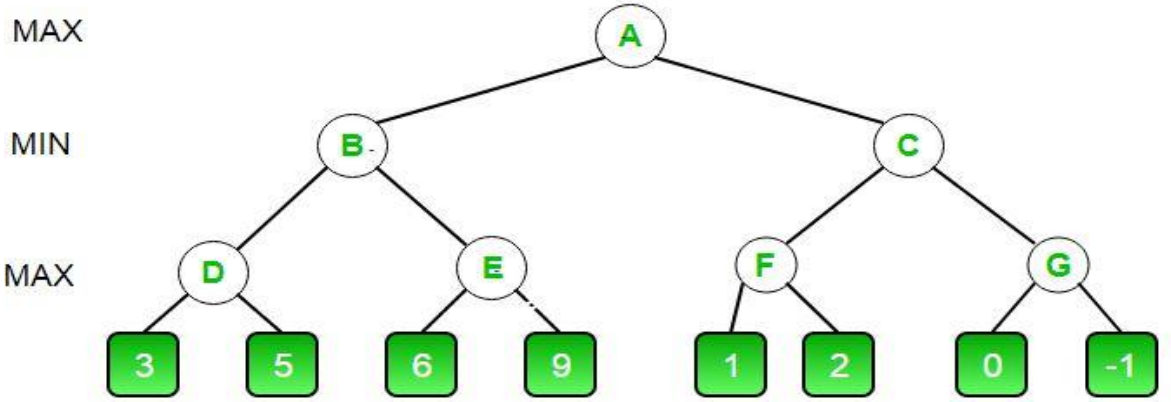
Example-2



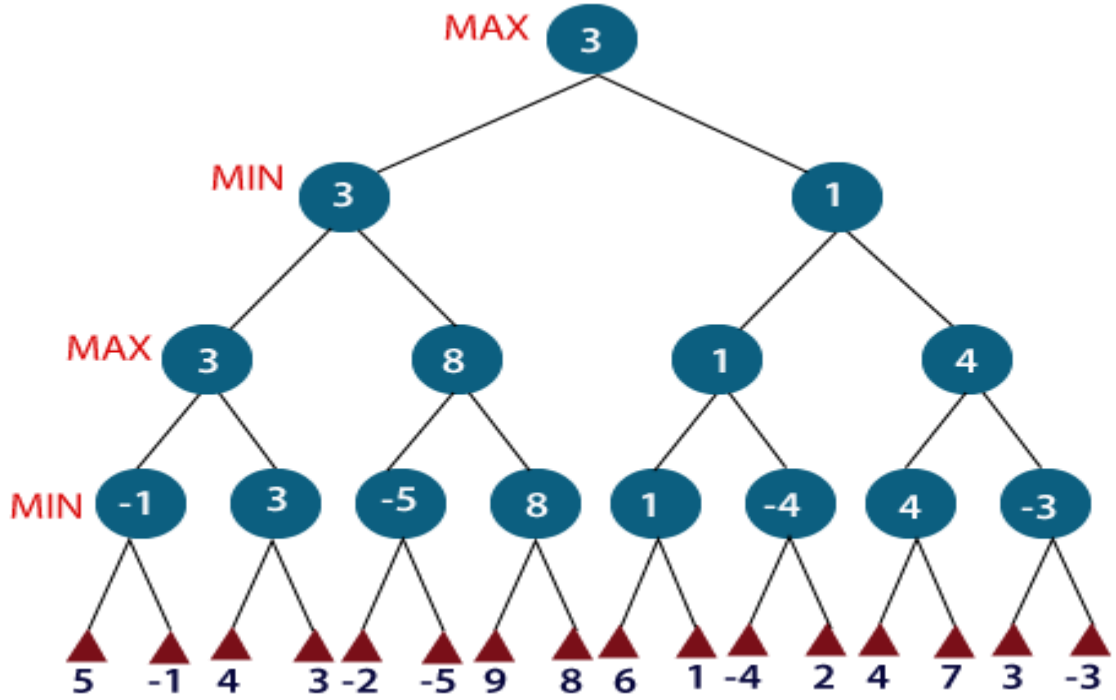
Example-3



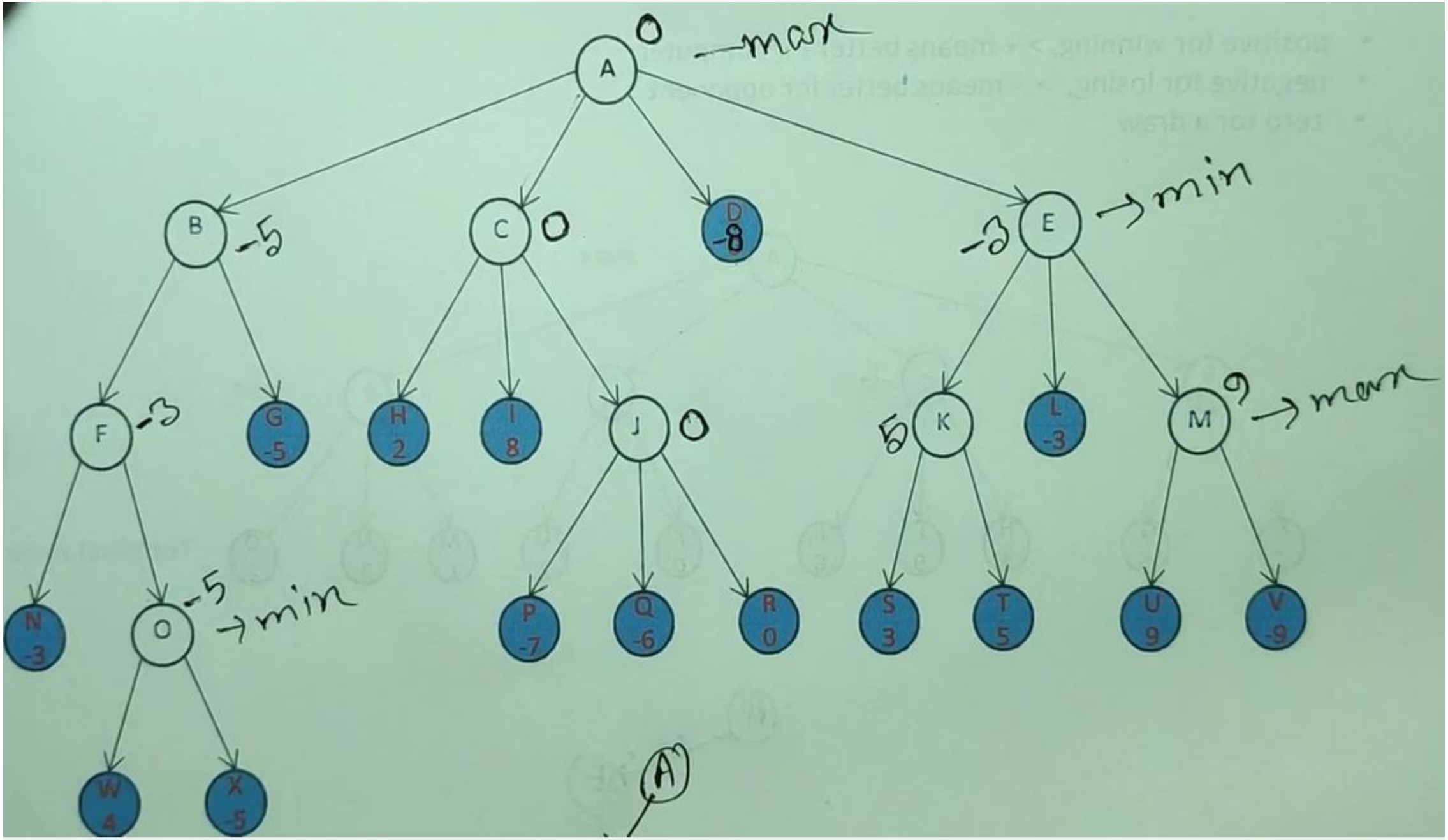
Example-4



Example-5



Example-6


A
0
max
B
-5
C
0
D
-8
E
-3
min
F
-3
G
-5
H
2
I
8
J
0
K
5
L
-3
M
9
max
N
-3
O
-5
min
P
-7
Q
-6
R
0
S
3
T
5
U
9
V
-9
W
4
X
-5
A

- **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)**.

- **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.

Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
  - **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
  - **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

- 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.

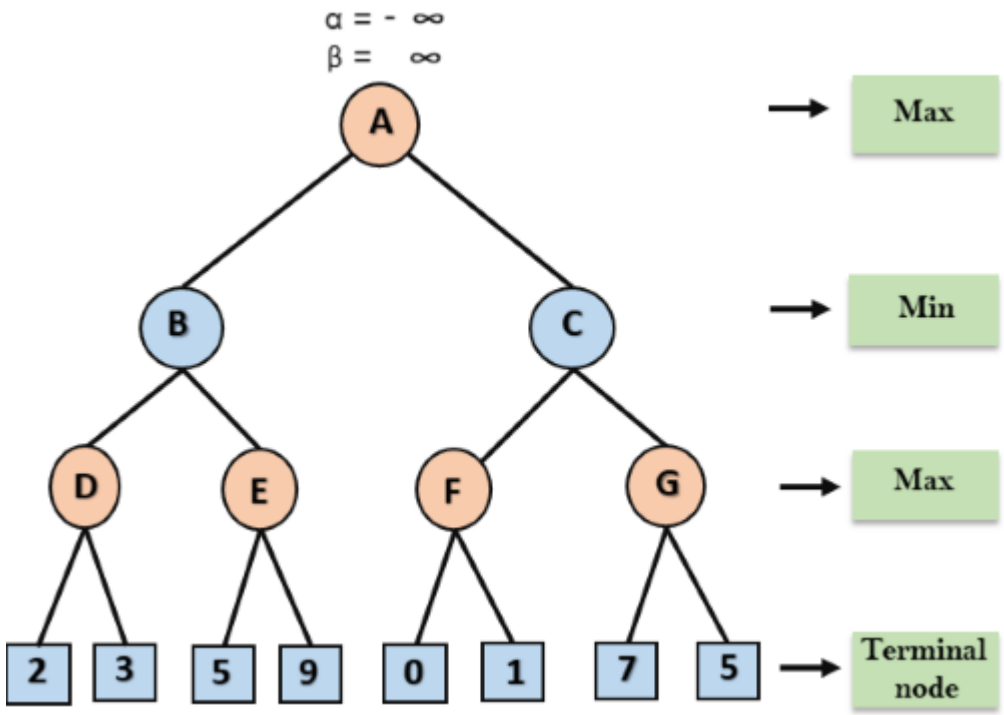
✓The Min player will only update the value of beta.

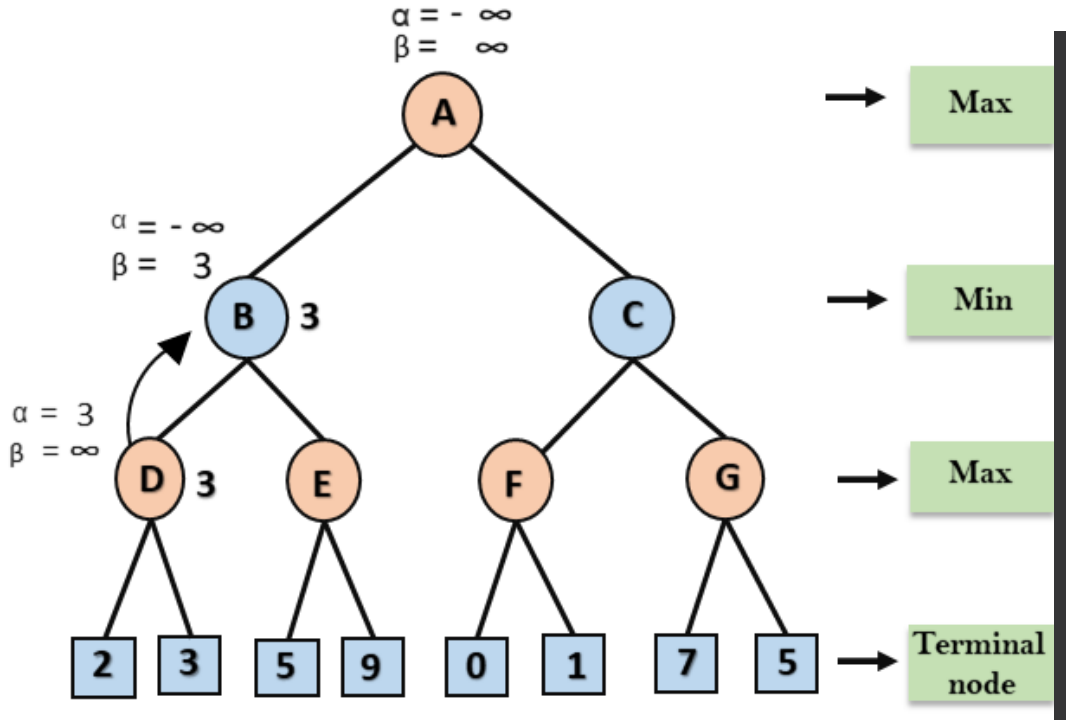
✓While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.

✓We will only pass the alpha, beta values to the child nodes.

Working of Alpha-Beta Pruning: example-1

- **Step 1:** At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.
- Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.
- Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.
- In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

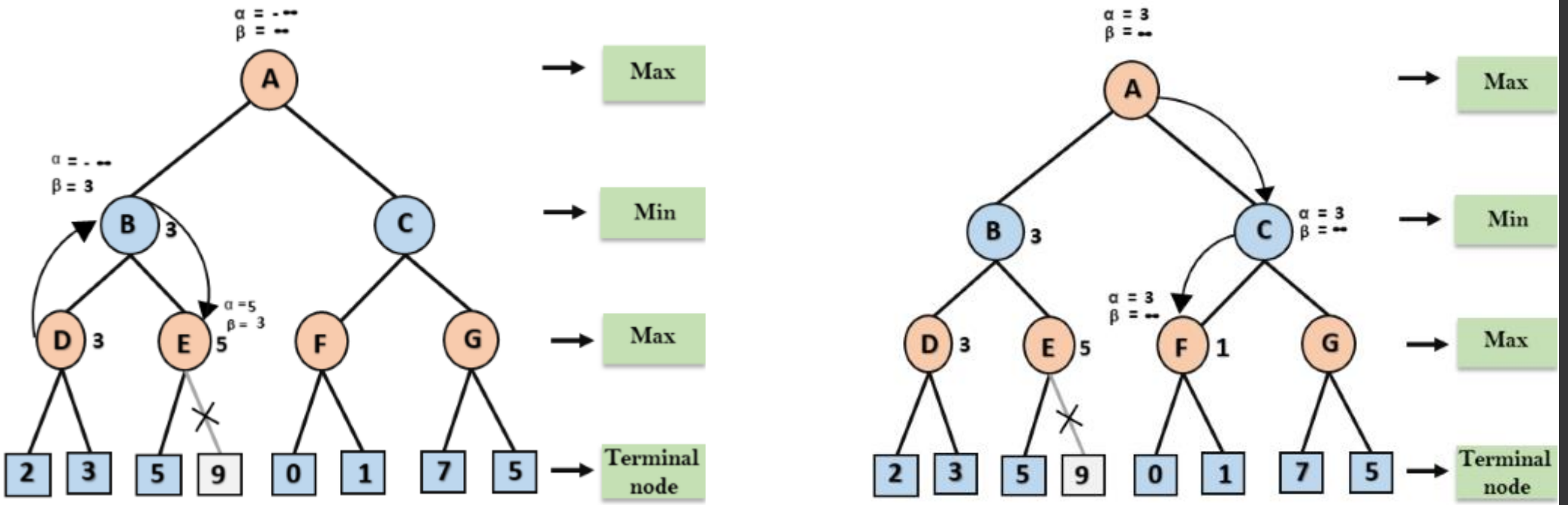




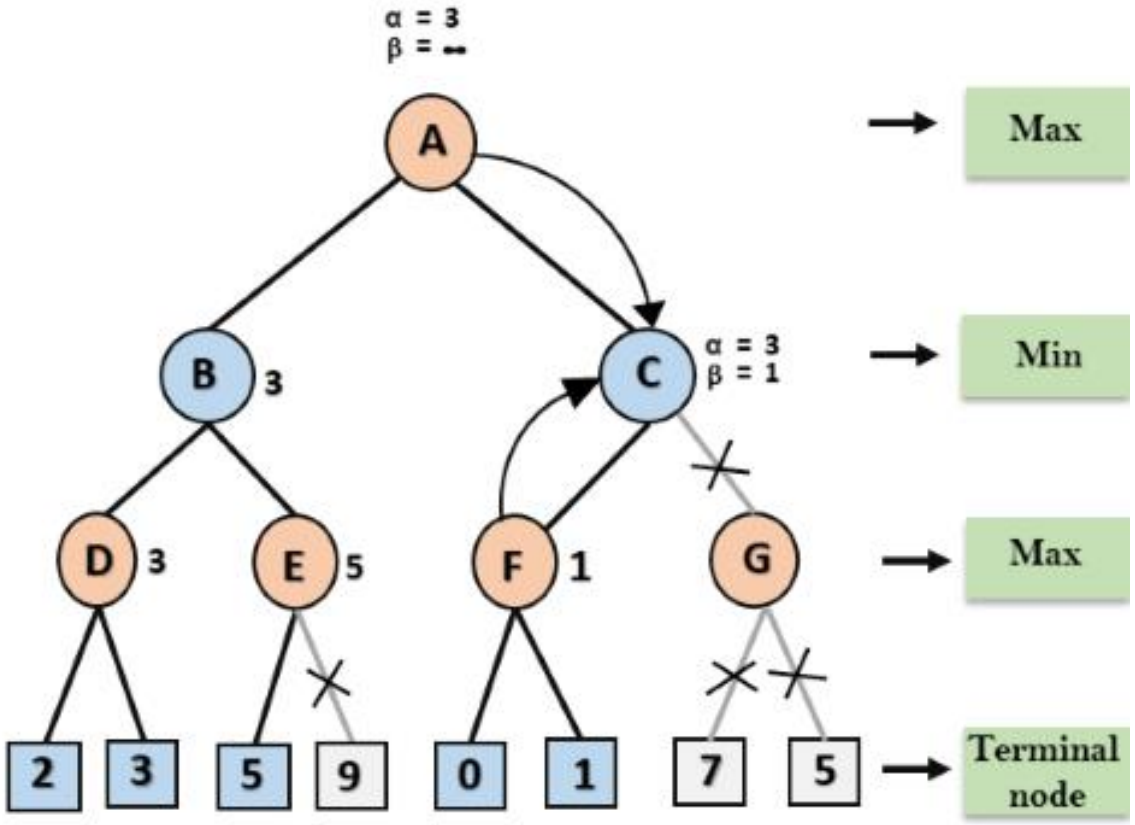
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.

Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
At node C, α=3 and β= +∞, and the same values will be passed on to node F.

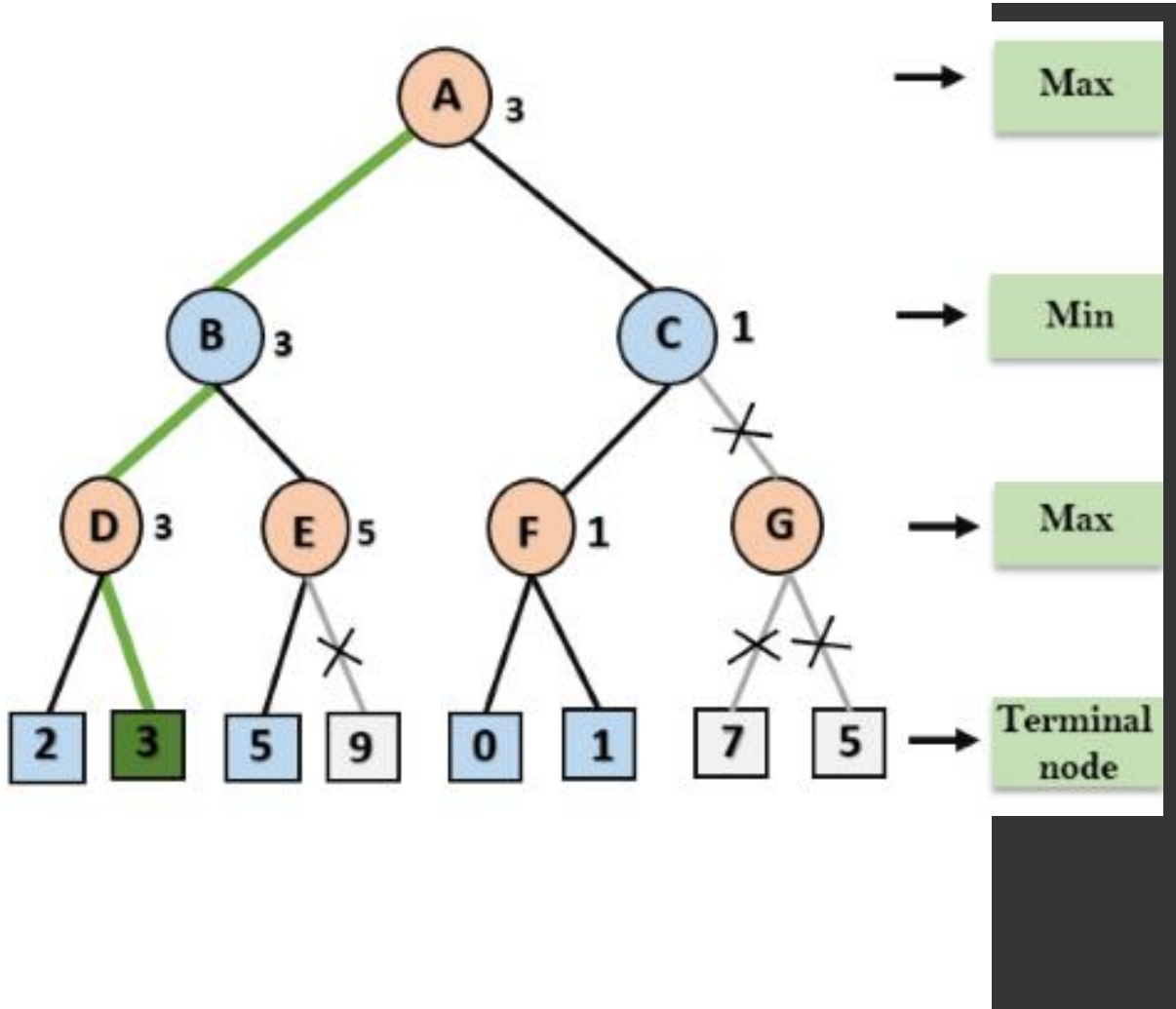
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



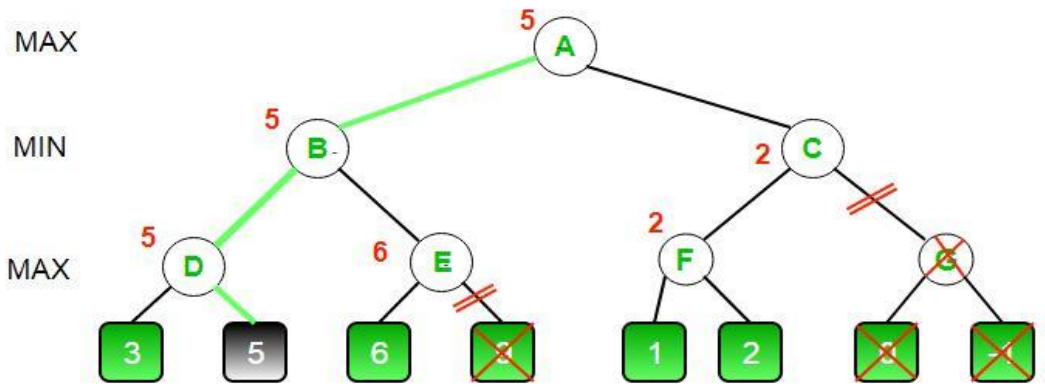
Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



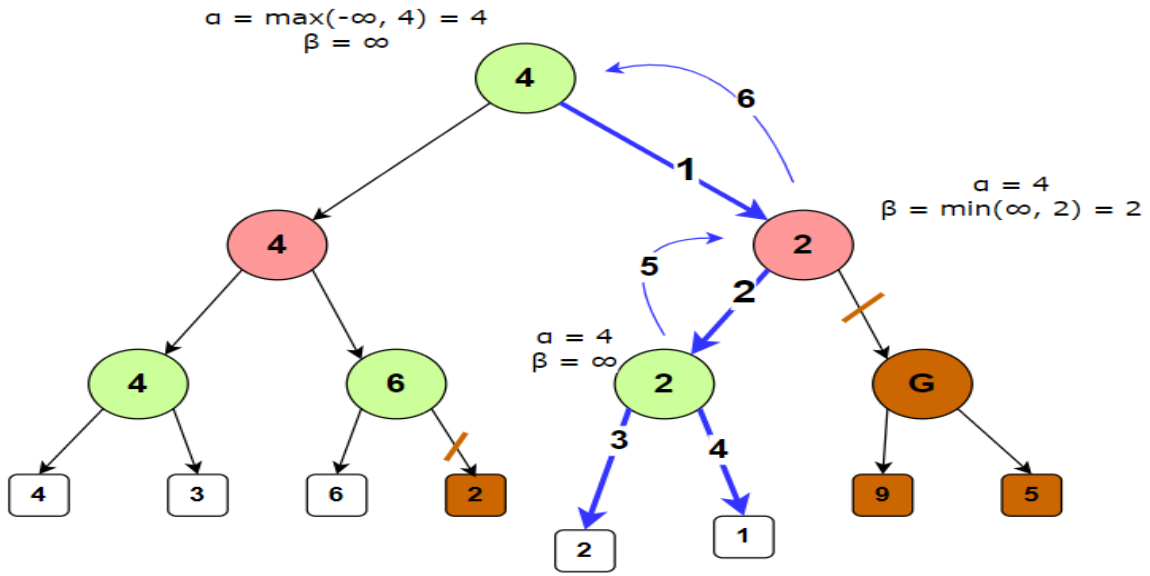
Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



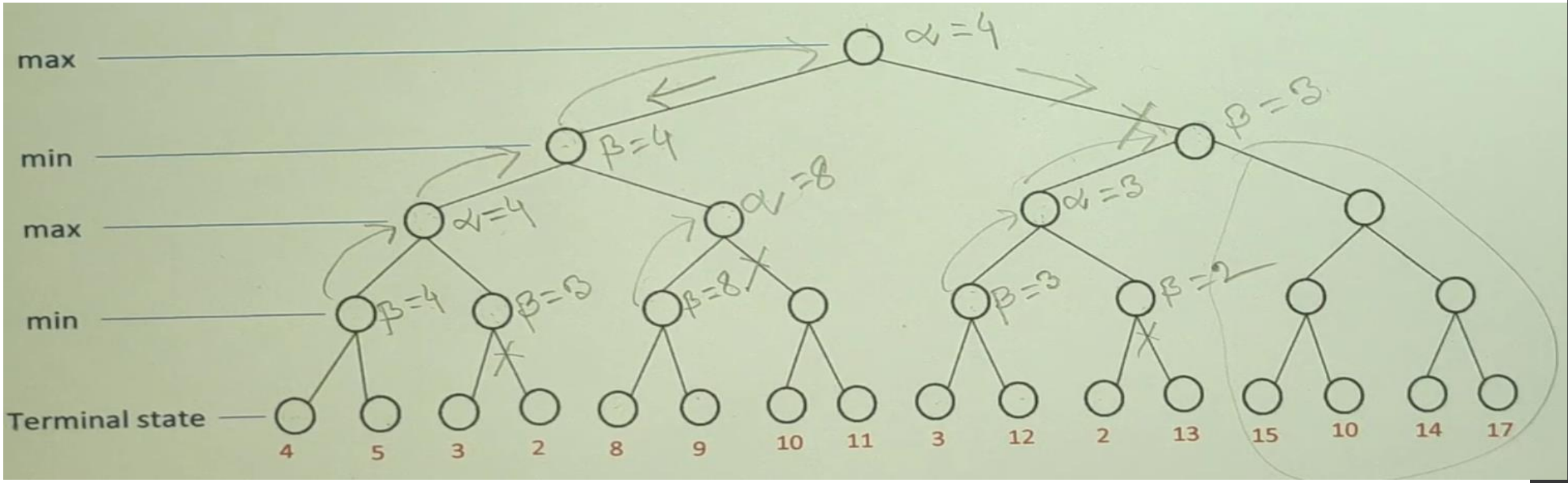
Example-2



Example-3

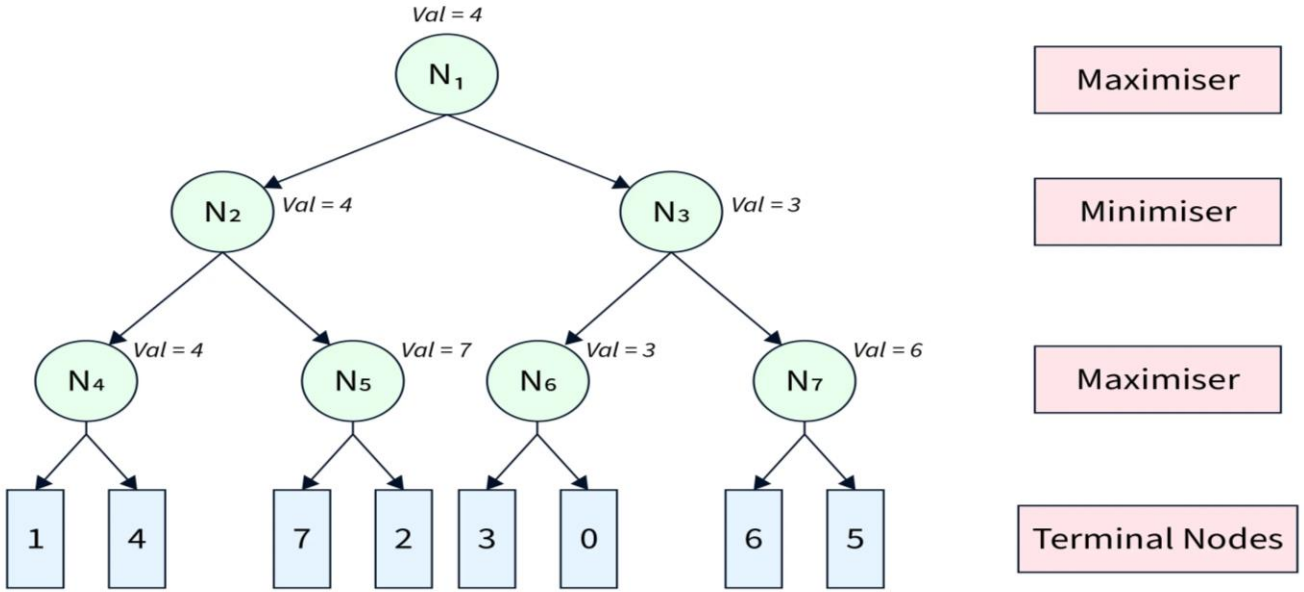


Example-4

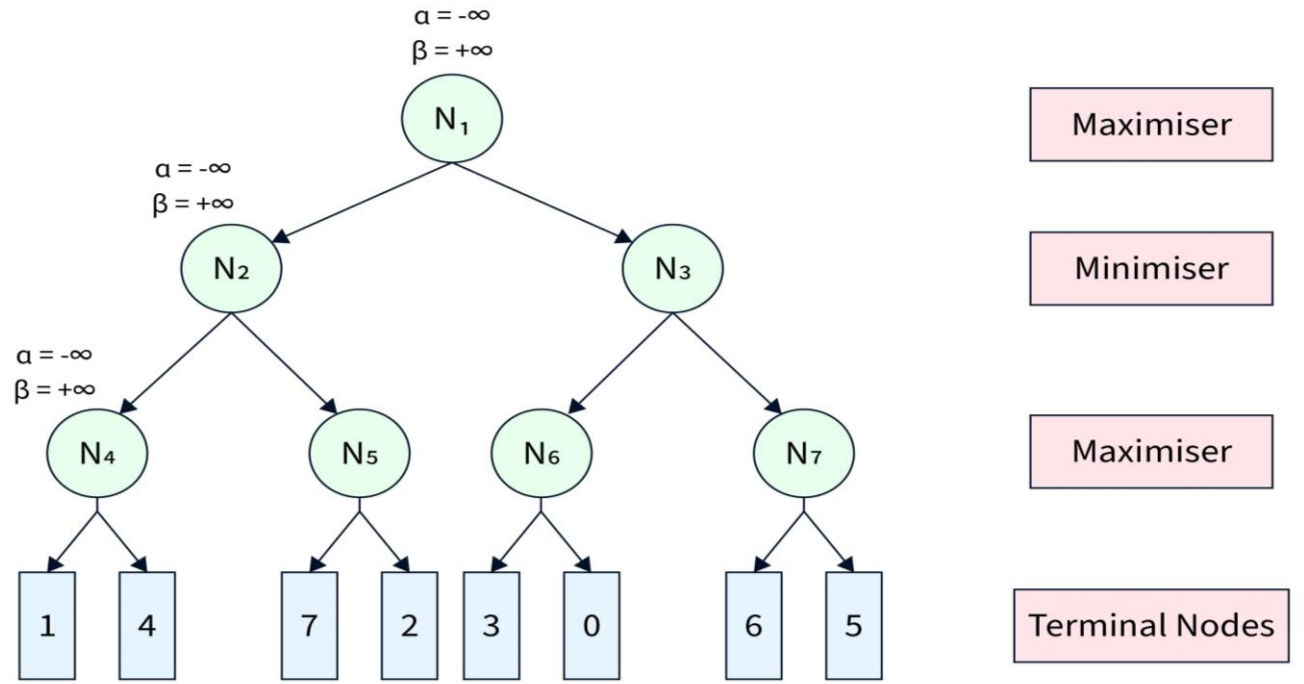


Example-5

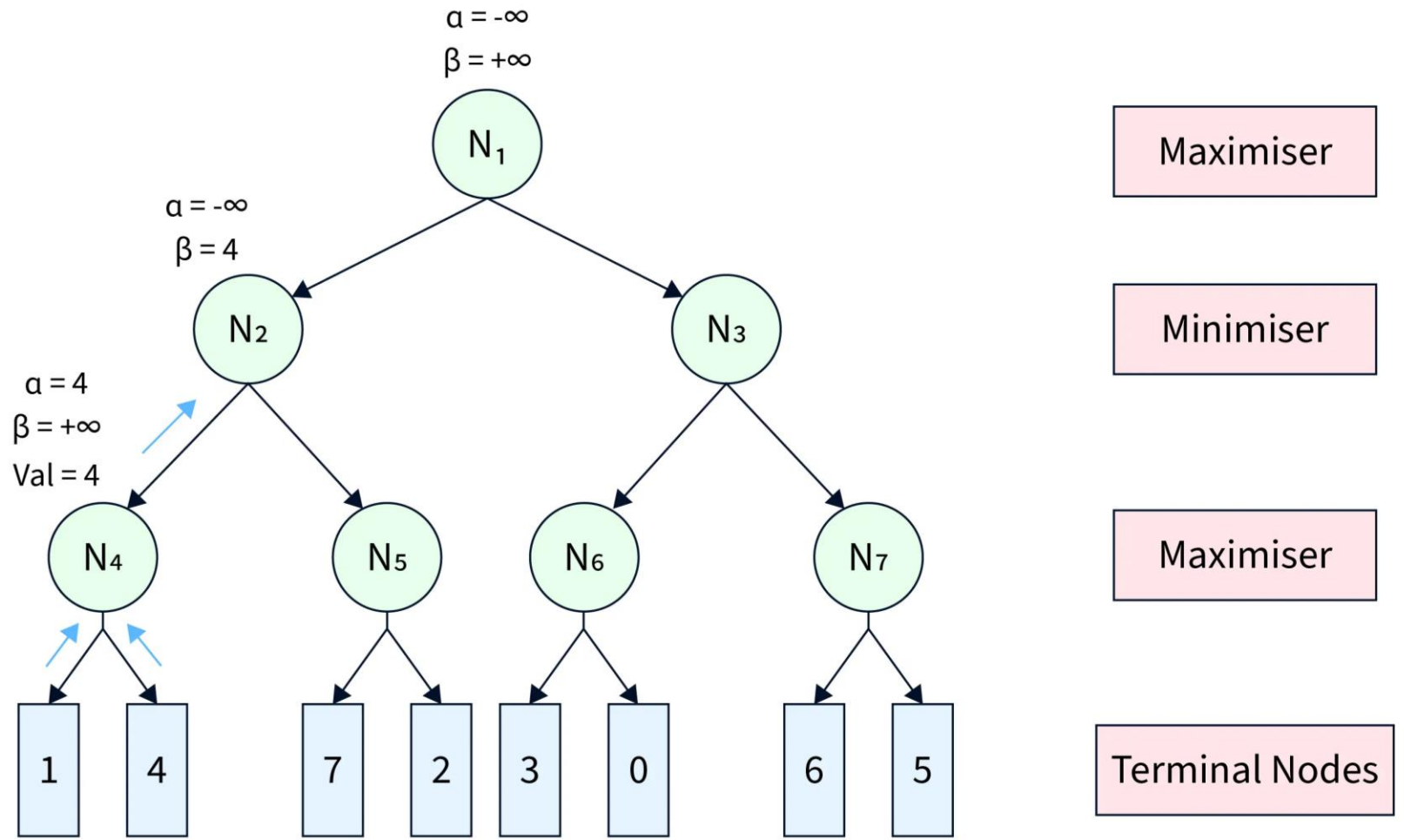
- The Minimax algorithm would have traversed the complete game tree leading to the game tree shown in Figure-2.



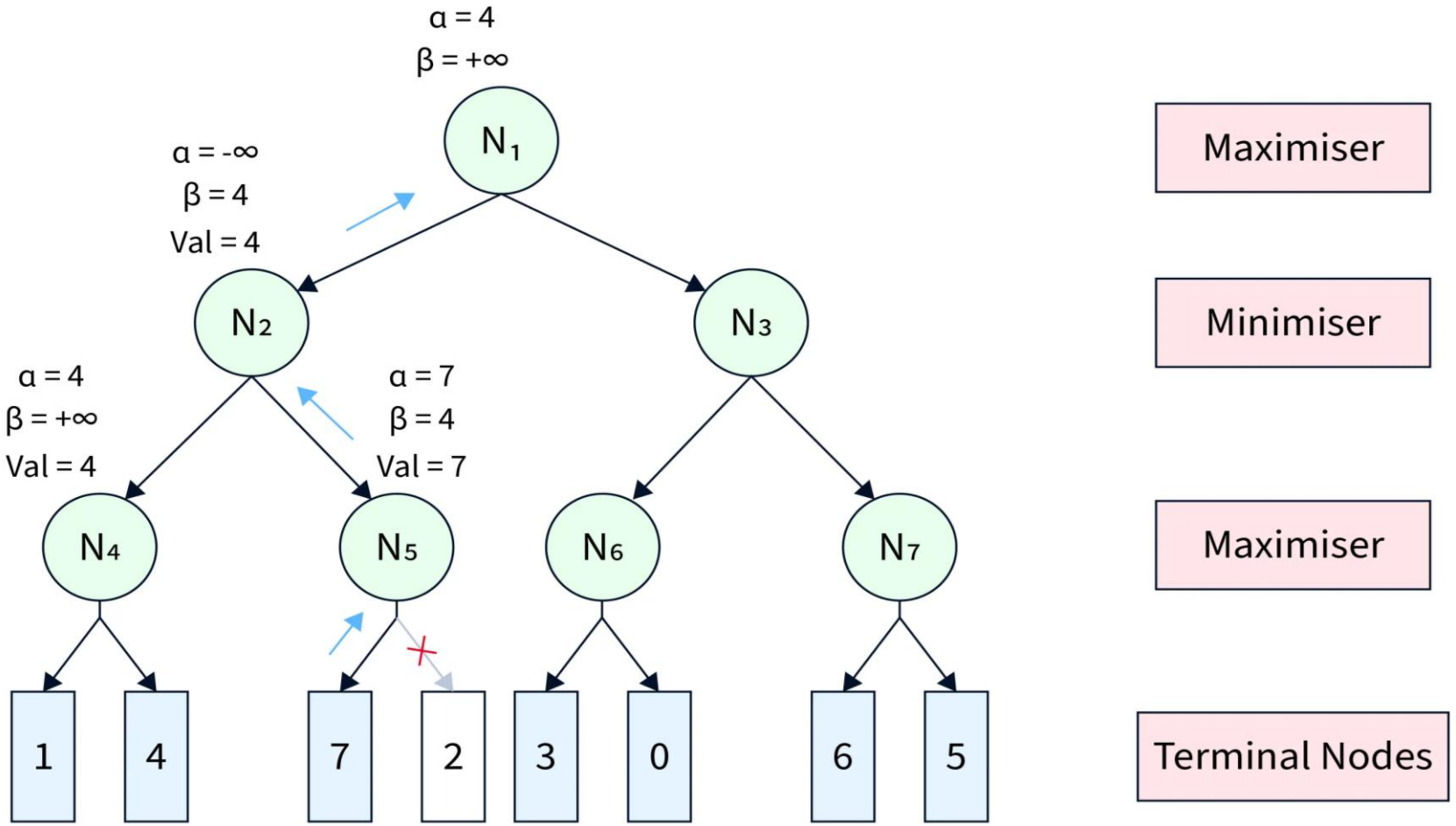
- Alpha Beta Pruning algorithm restricts the agent from visiting redundant sub-trees leading to faster average search time complexity. To visualize its working, we refer to the same game tree in Figure-1. We start from node $N1N1$ with α=−∞α=−∞ and β=∞β =∞. From $N1N1$, we move to its child node $N2N2$ with the same $\alpha\alpha$ and $\beta\beta$. Similarly, we move from $N2N2$ to its child $N4N4$, as shown in Figure-3.



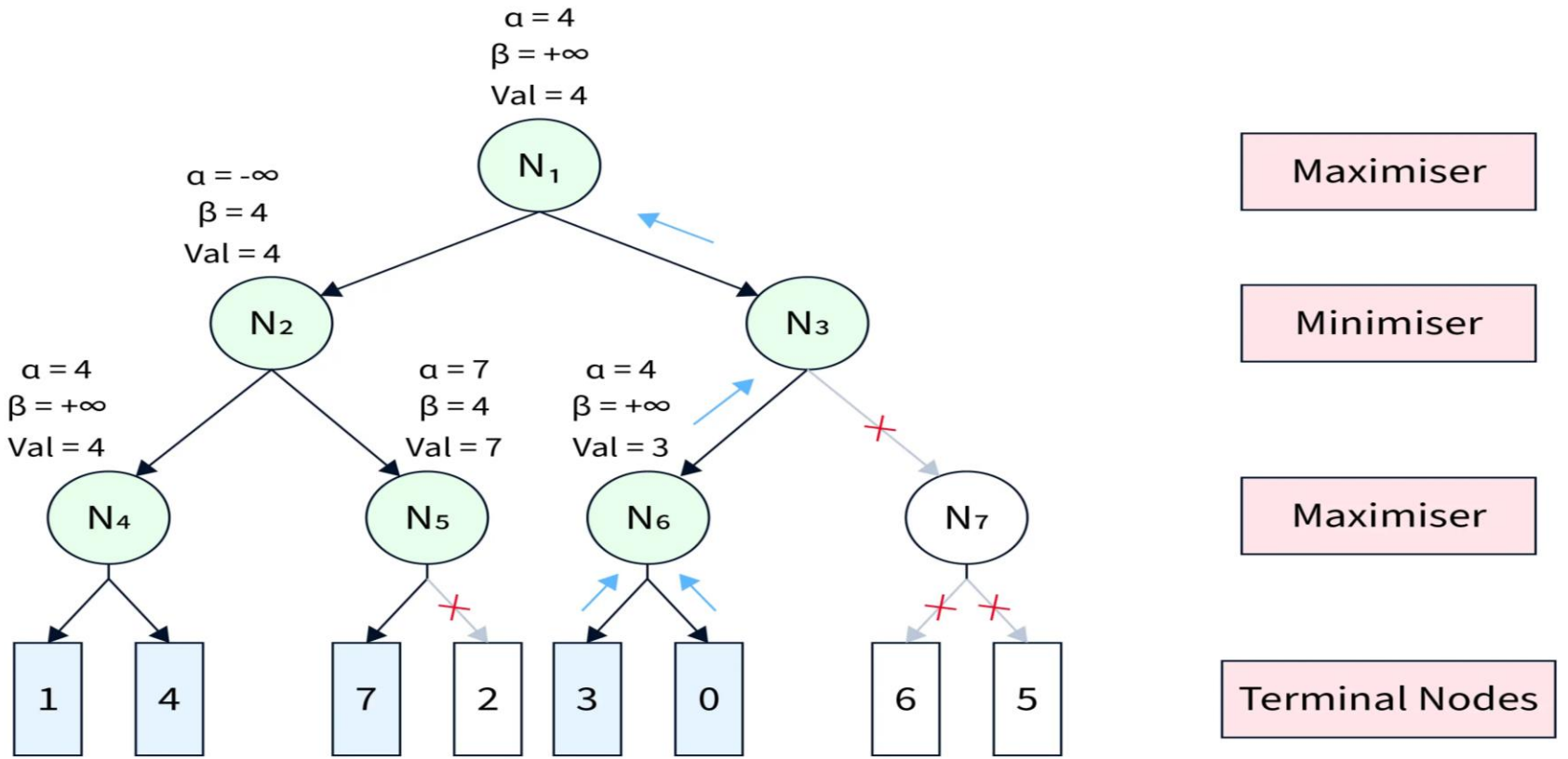
- From $N4N4$, we move to its first terminal child and get back its utility as 11. Since $N4N4$ is a maximiser, we update $\alpha=max(-\infty,1)=1\alpha=max(-\infty,1)=1$. The pruning condition $\alpha\geq\beta\alpha\geq\beta$ remains unsatisfied. Therefore, we move to the second terminal child of $N4N4$ and get its utility as 44. Further, we update $\alpha=max(1,4)=4\alpha=max(1,4)=4$.
- After this step, we get the node value of $N4N4$ as $max(1,4)=4max(1,4)=4$ and return it to its parent $N2N2$. Since $N2N2$ is a minimizer, it updates $\beta=min(+\infty,4)=4\beta=min(+\infty,4)=4$. The game tree after this step is shown in Figure-4.



- At $N2N2$, the pruning condition $\alpha \geq \beta \alpha \geq \beta$ is still unsatisfied. Therefore, we visit its next child node $N5N5$ with $\alpha = -\infty \alpha = -\infty$ and $\beta = 4 \beta = 4$. At $N5N5$, we get the utility of its first terminal child as 77. Since $N5N5$ is a maximiser, we update $\alpha = max(-\infty, 7) = 7 \alpha = max(-\infty, 7) = 7$. Now, the pruning condition gets satisfied $(7 \geq 4)(7 \geq 4)$, leading to the pruning of the other child node. After that, the node utility of $N5 = 7 N5 = 7$ is further sent back to the parent node $N2N2$.
- $N2N2$ updates $\beta = min(4,7) = 4 \beta = min(4,7) = 4$. Further, the node value of $N2 = min(4,7) = 4 N2 = min(4,7) = 4$ is returned to its parent $N1N1$. At $N1N1$, we update $\alpha = max(-\infty, 4) = 4 \alpha = max(-\infty, 4) = 4$. After this step, the game tree looks like Figure-5.



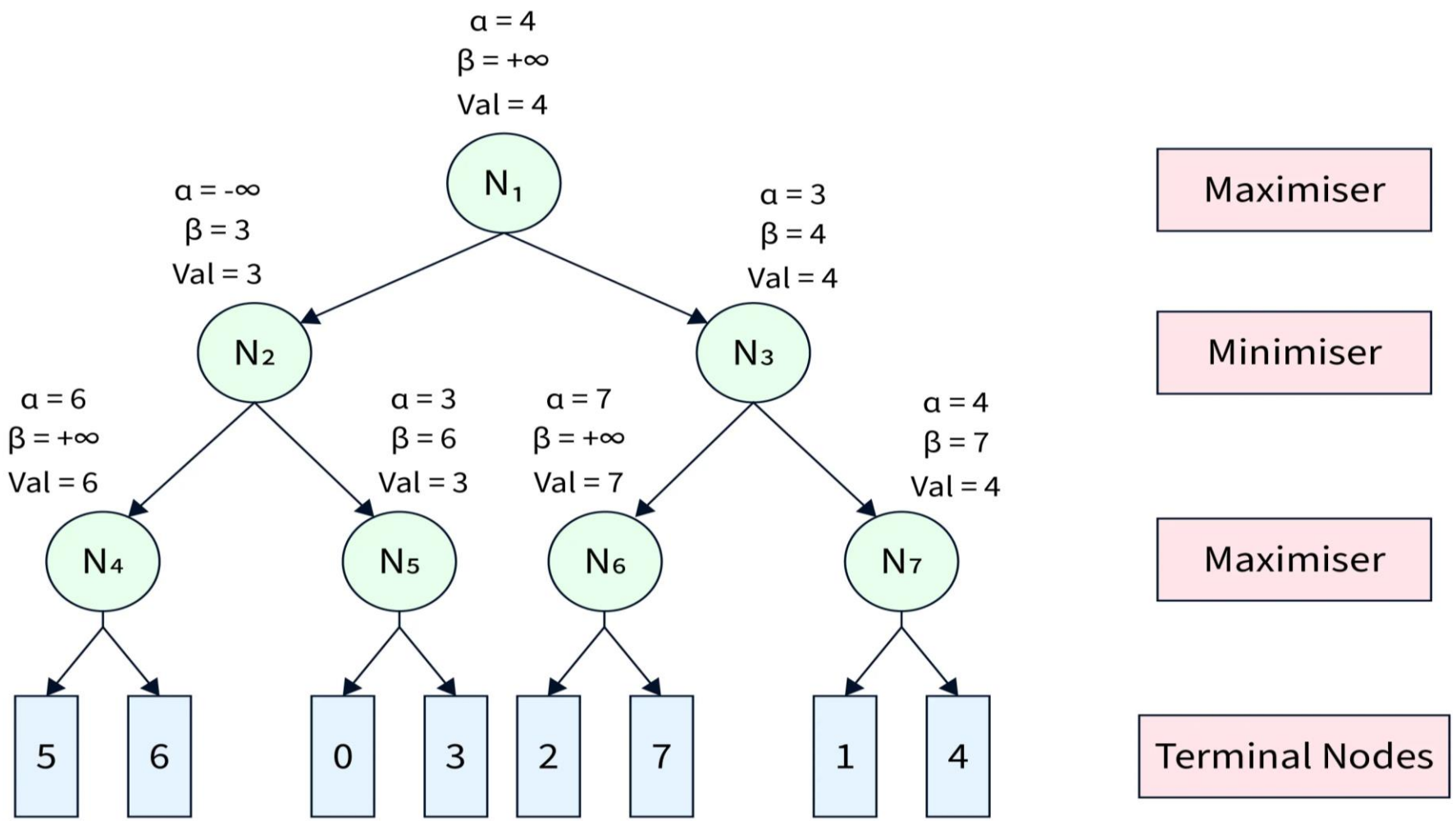
- The pruning condition is still unsatisfactory at $N1N1$. Therefore, we continue towards the next child $N3N3$, with the same values of $\alpha\alpha$ and $\beta\beta$. Similarly, we propagate from $N3N3$ to its first child $N6N6$. At $N6N6$, we get back the utility of its first terminal child as 33 and update $\alpha=max(4,3)=4\alpha=max(4,3)=4$. Since the pruning condition is still unsatisfied, we get the utility of the second terminal child as 00 and update $\alpha=max(4,0)=4\alpha=max(4,0)=4$. After this, the node value of $N6=max(3,0)=3N6=max(3,0)=3$, is returned to the parent $N3N3$. Since $N3N3$ is a minimizer, we update $\beta=min(+\infty,3)=3\beta=min(+\infty,3)=3$.
- Now, the pruning condition gets satisfied $(4\geq3)(4\geq3)$. Therefore, the other child sub-tree(rooted at $N7N7$) of $N3N3$ is pruned, and the node value of $N3N3$ becomes 33, which is returned to parent $N1N1$. At $N1N1$, we again update $\alpha=max(4,3)=4\alpha=max(4,3)=4$ and the final utility of $N1N1$ becomes 44. All these steps are shown in Figure-6.



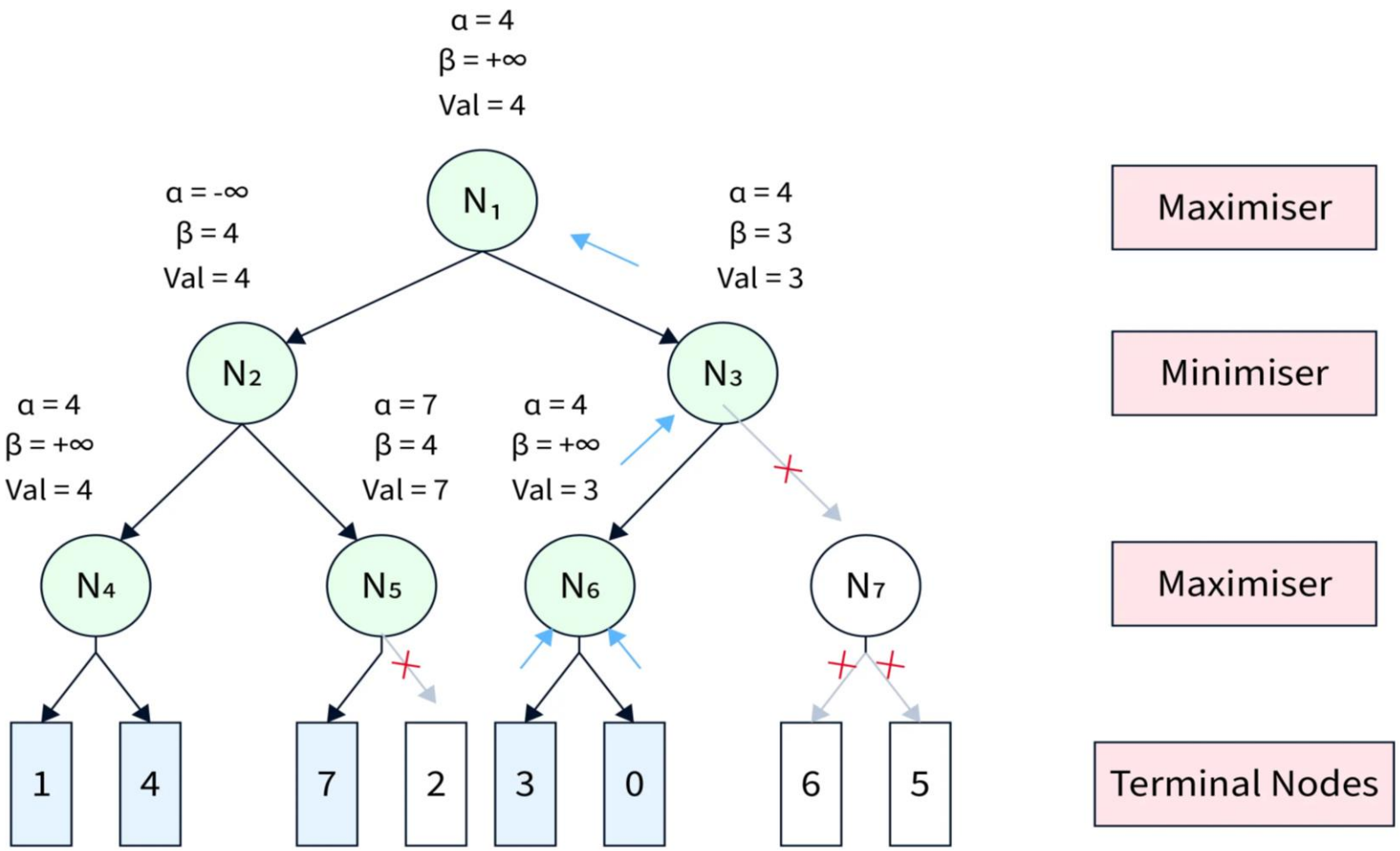
Move Ordering in Alpha Beta pruning in AI

The performance of the Alpha Beta Pruning algorithm is highly dependent on the order in which the nodes are traversed. For instance,

Case-1(Worst Performance): If the best sequence of actions is on the right of the game tree, then there will be no pruning, and all the nodes will be traversed, leading to even worse performance than the Minimax algorithm because of the overhead of computing $\alpha\alpha$ and $\beta\beta$. An example game tree can be seen in Figure-7.



Case-2(Best Performance): If the best sequence of actions is on the left of the game tree, there will be a lot of pruning, and only about half of the nodes will be traversed to get the optimal result.
An example game tree can be seen in Figure-8.



Rules to Find Good Ordering

- The best move occurs from the shallowest node.
- The game tree is ordered so that the best nodes are checked first.
- Domain knowledge is used to find the best move.
- Memoization is applied to prevent recomputation in case of repeating states.

- **Advantages of Game Playing in Artificial Intelligence:**

  1.Advancement of AI: Game playing has been a driving force behind the development of artificial intelligence and has led to the creation of new algorithms and techniques that can be applied to other areas of AI.

  2.Education and training: Game playing can be used to teach AI techniques and algorithms to students and professionals, as well as to provide training for military and emergency response personnel.

  3.Research: Game playing is an active area of research in AI and provides an opportunity to study and develop new techniques for decision-making and problem-solving.

  4.Real-world applications: The techniques and algorithms developed for game playing can be applied to real-world applications, such as robotics, autonomous systems, and decision support systems.

- **Disadvantages of Game Playing in Artificial Intelligence:**

  1.Limited scope: The techniques and algorithms developed for game playing may not be well-suited for other types of applications and may need to be adapted or modified for different domains.

  2.Computational cost: Game playing can be computationally expensive, especially for complex games such as chess or Go, and may require powerful computers to achieve real-time performance.