

# Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning)

Prerequisites: [Minimax Algorithm in Game Theory](#), [Evaluation Function in Game Theory](#)

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.

**Alpha** is the best value that the **maximizer** currently can guarantee at that level or above.

**Beta** is the best value that the **minimizer** currently can guarantee at that level or above.

**Pseudocode :**

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

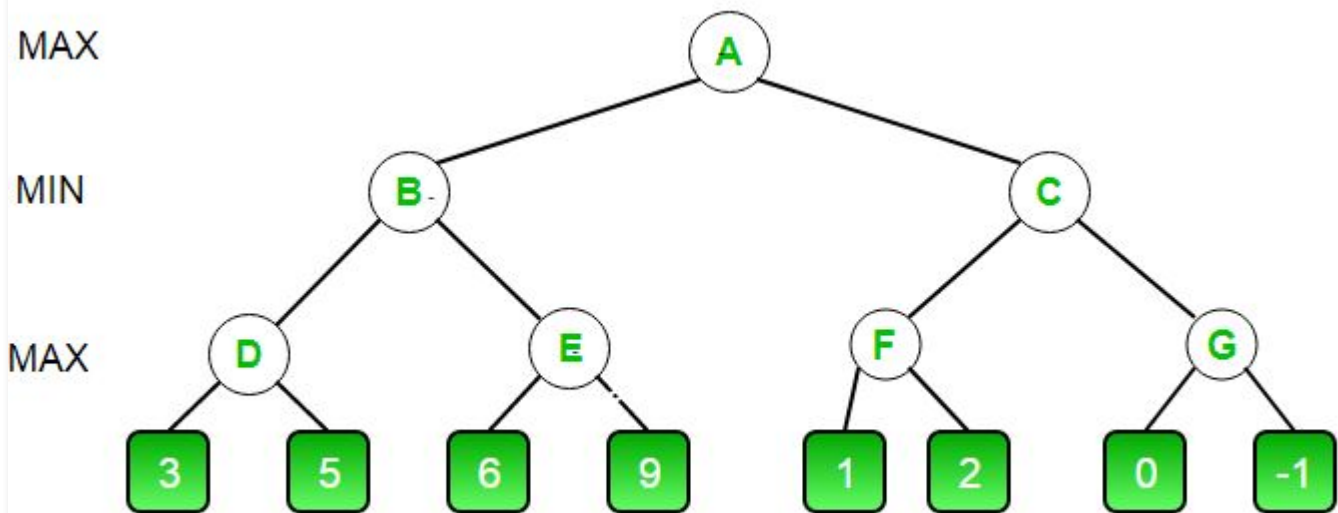
    if node is a leaf node :
        return value of the node

    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

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

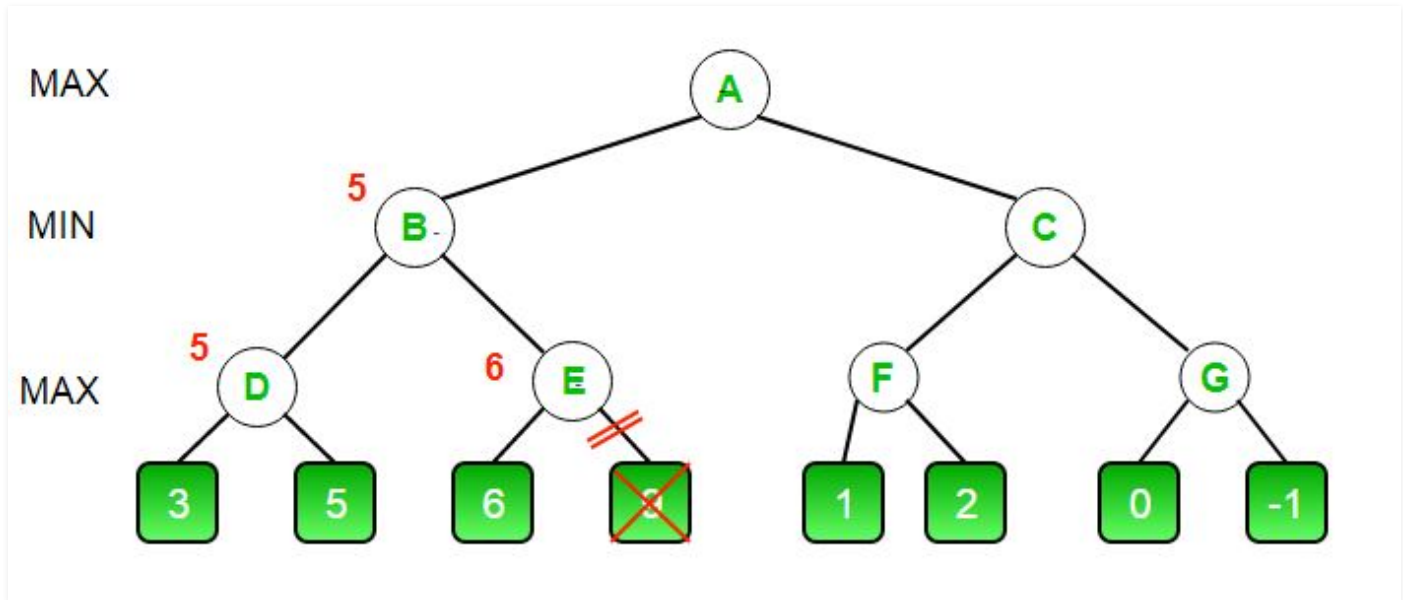
```
// Calling the function for the first time.
minimax(0, 0, true, -INFINITY, +INFINITY)
```

Let's make above algorithm clear with an example.



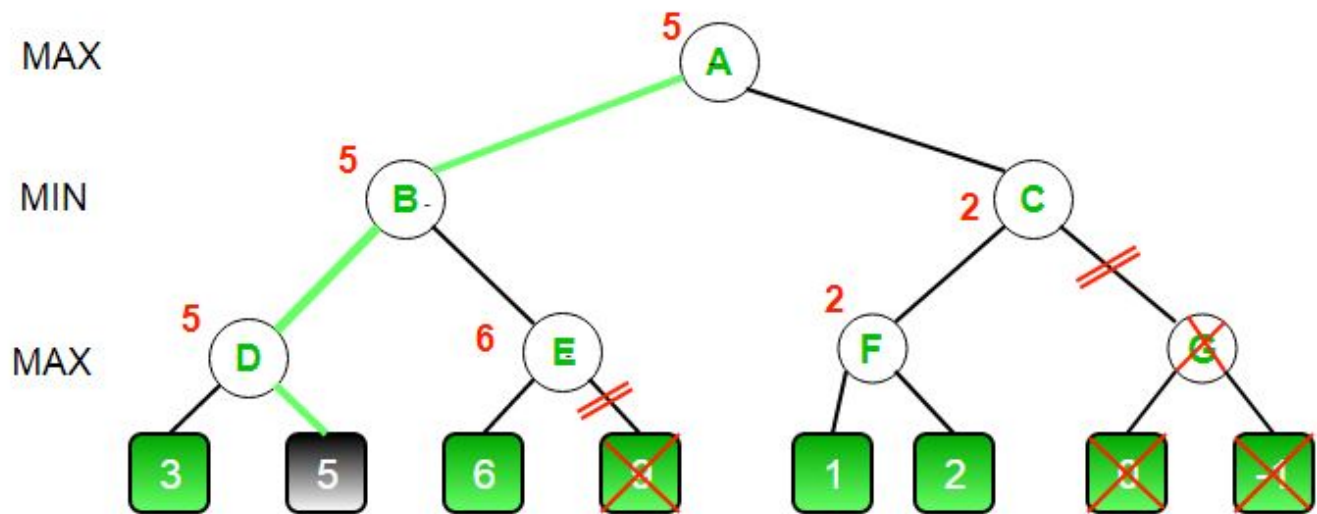
- The initial call starts from **A**. The value of alpha here is **-INFINITY** and the value of beta is **+INFINITY**. These values are passed down to subsequent nodes in the tree. At **A** the maximizer must choose max of **B** and **C**, so **A** calls **B** first
- At **B** it the minimizer must choose min of **D** and **E** and hence calls **D** first.
- At **D**, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at **D** is  $\max(-\text{INF}, 3)$  which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition  $\beta \leq \alpha$ . This is false since  $\beta = +\text{INF}$  and  $\alpha = 3$ . So it continues the search.
- **D** now looks at its right child which returns a value of 5. At **D**,  $\alpha = \max(3, 5)$  which is 5. Now the value of node **D** is 5
- **D** returns a value of 5 to **B**. At **B**,  $\beta = \min(+\text{INF}, 5)$  which is 5. The minimizer is now guaranteed a value of 5 or lesser. **B** now calls **E** to see if he can get a lower value than 5.
- At **E** the values of alpha and beta is not  $-\text{INF}$  and  $+\text{INF}$  but instead  $-\text{INF}$  and 5 respectively, because the value of beta was changed at **B** and that is what **B** passed down to **E**
- Now **E** looks at its left child which is 6. At **E**,  $\alpha = \max(-\text{INF}, 6)$  which is 6. Here the condition becomes true.  $\beta$  is 5 and  $\alpha$  is 6. So  $\beta \leq \alpha$  is true. Hence it breaks and **E** returns 6 to **B**
- Note how it did not matter what the value of **E**'s right child is. It could have been  $+\text{INF}$  or  $-\text{INF}$ , it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of **B**. This way we dint have to look at that 9 and hence saved computation time.
- **E** returns a value of 6 to **B**. At **B**,  $\beta = \min(5, 6)$  which is 5. The value of node **B** is also 5

So far this is how our game tree looks. The 9 is crossed out because it was never computed.



- **B** returns 5 to **A**. At **A**,  $\alpha = \max(-\text{INF}, 5)$  which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**,  $\alpha = 5$  and  $\beta = +\text{INF}$ . **C** calls **F**
- At **F**,  $\alpha = 5$  and  $\beta = +\text{INF}$ . **F** looks at its left child which is a 1.  $\alpha = \max(5, 1)$  which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**,  $\beta = \min(+\text{INF}, 2)$ . The condition  $\beta \leq \alpha$  becomes false as  $\beta = 2$  and  $\alpha = 5$ . So it breaks and it does not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is  $\max(5, 2)$  which is a 5.
- Hence the optimal value that the maximizer can get is 5

This is how our final game tree looks like. As you can see **G** has been crossed out as it was never computed.



## C++

```
// C++ program to demonstrate
// working of Alpha-Beta Pruning
#include<bits/stdc++.h>
using namespace std;

// Initial values of
// Alpha and Beta
const int MAX = 1000;
const int MIN = -1000;

// Returns optimal value for
// current player(Initially called
// for root and maximizer)
int minimax(int depth, int nodeIndex,
            bool maximizingPlayer,
            int values[], int alpha,
            int beta)
{
    // Terminating condition. i.e
    // leaf node is reached
    if (depth == 3)
        return values[nodeIndex];

    if (maximizingPlayer)
    {
        int best = MIN;

        // Recur for left and
        // right children
        for (int i = 0; i < 2; i++)
        {
            int val = minimax(depth + 1, nodeIndex * 2 + i,
                              false, values, alpha, beta);
            best = max(best, val);
            alpha = max(alpha, best);

            // Alpha Beta Pruning
            if (beta <= alpha)
                break;
        }
        return best;
    }
}
```

```
}
else
{
    int best = MAX;

    // Recur for left and
    // right children
    for (int i = 0; i < 2; i++)
    {
        int val = minimax(depth + 1, nodeIndex * 2 + i,
                           true, values, alpha, beta);
        best = min(best, val);
        beta = min(beta, best);

        // Alpha Beta Pruning
        if (beta <= alpha)
            break;
    }
    return best;
}
}

// Driver Code
int main()
{
    int values[8] = { 3, 5, 6, 9, 1, 2, 0, -1 };
    cout <<"The optimal value is : "<< minimax(0, 0, true, val
    return 0;
}
```

[Run on IDE](#)

Java

**Output :**

The optimal value is : 5

This article is contributed by **Akshay L Aradhya**. If you like GeeksforGeeks and would like to contribute, you can also write an article using [contribute.geeksforgeeks.org](https://contribute.geeksforgeeks.org) or mail your article to [contribute@geeksforgeeks.org](mailto:contribute@geeksforgeeks.org). See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**Improved By :** [vinayb21](#), [Akshay Aradhya](#)

Practice Tags : [Game Theory](#)

Article Tags : [Game Theory](#)



Please write to us at [contribute@geeksforgeeks.org](mailto:contribute@geeksforgeeks.org) to report any issue with the above [Login to Improve this Article](#) content.

## Recommended Posts:

[Minimax Algorithm in Game Theory | Set 1 \(Introduction\)](#)  
[Minimax Algorithm in Game Theory | Set 3 \(Tic-Tac-Toe AI – Finding optimal move\)](#)  
[Minimax Algorithm in Game Theory | Set 5 \(Zobrist Hashing\)](#)  
[Minimax Algorithm in Game Theory | Set 2 \(Introduction to Evaluation Function\)](#)  
[Implementation of Tic-Tac-Toe game](#)  
[Josephus Problem | \(Iterative Solution\)](#)  
[Find the winner in nim-game](#)  
[Game of Nim with removal of one stone allowed](#)  
[Check if the game is valid or not](#)  
[Variation in Nim Game](#)