Min-Max Algorithm and Pruning

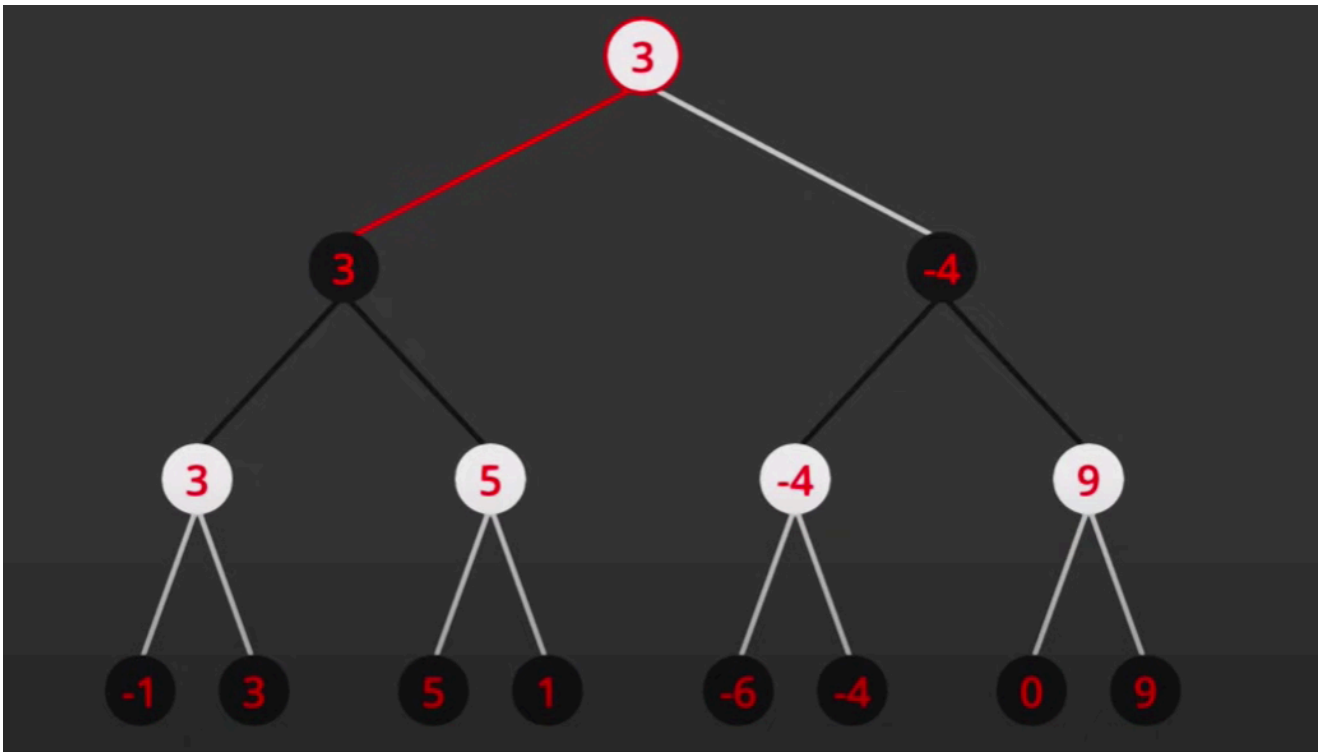
The Min-Max algorithm is a recursive / backtracking algorithm which is used for decision making in games . It used DFS to search the game tree for the best possible moves. It is mostly used in AI for games like chess , tic tac toe. This algorithm computes the minimax decision for the current state.

Working of Minimax

Lets understand this by taking an example of chess, chess is a two player game and there are two players one being the Maximizer (white pieces) and one being the minimizer (black pieces). In this scenario the maximizer will try to get the maximum value while the minimizer will try to achieve the minimum value.

Higher number means advantage for white pieces (maximizer) and lower number means advantage for the black pieces (minimizer).

The algorithm applies DFS (Depth First Search) on the entire game tree so we have to reach the terminal nodes and then at the terminal nodes we compare the value and based on whose turn it is we return the best possible value (best possible move).



In the above example at the first level level black chooses 3 between 3 and 5 as it wants to minimize its value to get an advantage while at the second level white chooses 3 b/w -1 and 3 because it wants to maximize its value to get an advantage.

Code

```
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)
```

Step 1 - The function minimax takes 3 variables

1. Position - which tells us the current position

2. depth - which defines how deep we want to search
3. maximizing player - The white pieces

Step 2 - We start by checking if the depth is zero the start of the game tree or if the game is over, if yes we return the static evaluation of the position.

Step 3 - Now comes the turn of the maximizing player (since white moves first in chess). we want to find the highest possible evaluation from this position. we declare a variable called `#maxEval` this variable will store the maximum evaluation for white piece and its initial value is set to - infinity which is the worst possible value it can have. Then we traverse through the tree and check possible child node till we reach the terminal node. For defining the evaluation of each child we define a variable called `#eval` and make a recursive call to the minimax function passing the child node and the depth). Now we can set max eval to the value which is greater b/w `#maxEval` and `#eval` and return the maximum evaluation .

Step 4 - Now its the turn of the black pieces and we do the same here except we want the minimum possible evaluation , so we define a variable called `#minEval` and compare it with the value of all the evaluation and after comparing set the value of `#minEval` to the lowest possible evaluation and then return the minimum evaluation.

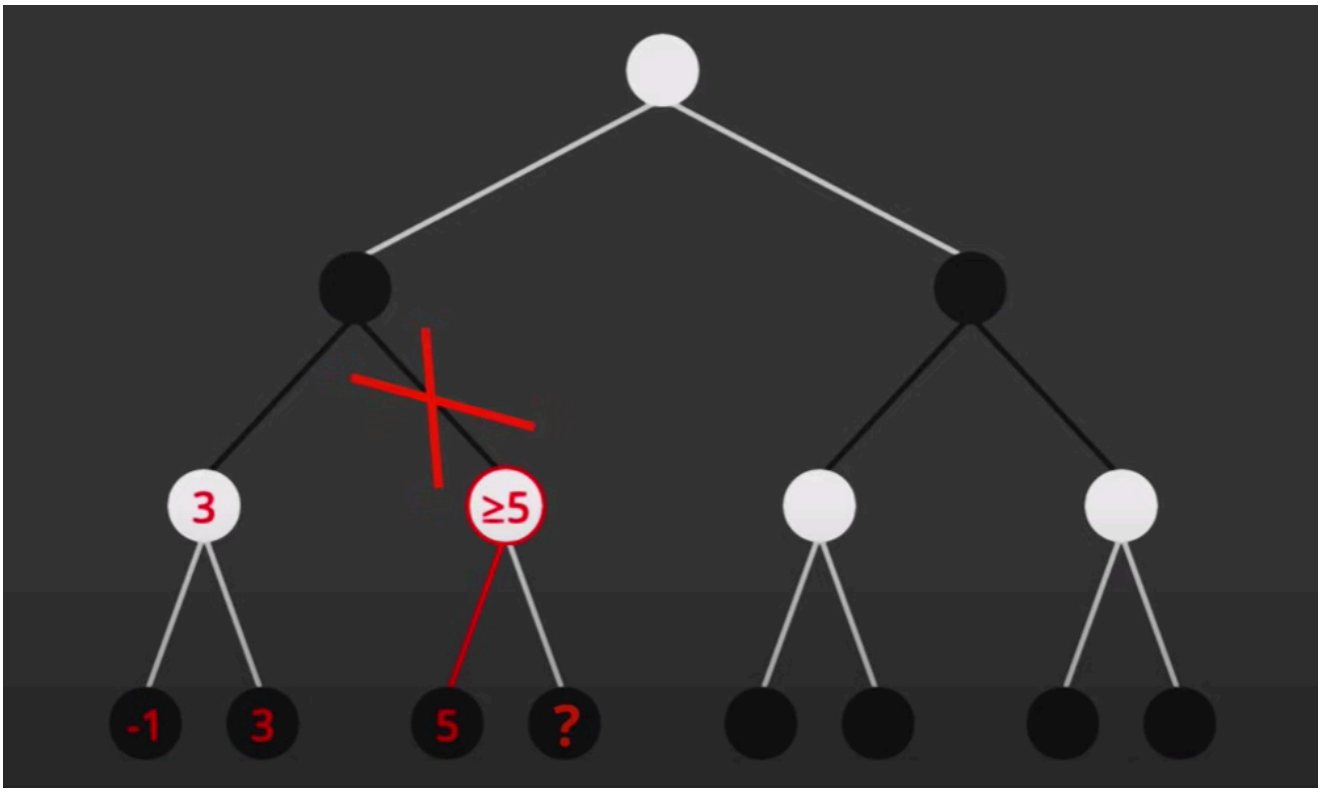
Alpha - Beta Pruning

Alpha Beta pruning is a optimization technique for the min max algorithm . Since the min max algorithm uses DFS search it has to search through every node which sometimes can take a lot of time and computational power. With this technique we can find the best possible move without checking each node . This technique involves two threshold parameters Alpha and Beta that's why it is called alpha beta pruning.

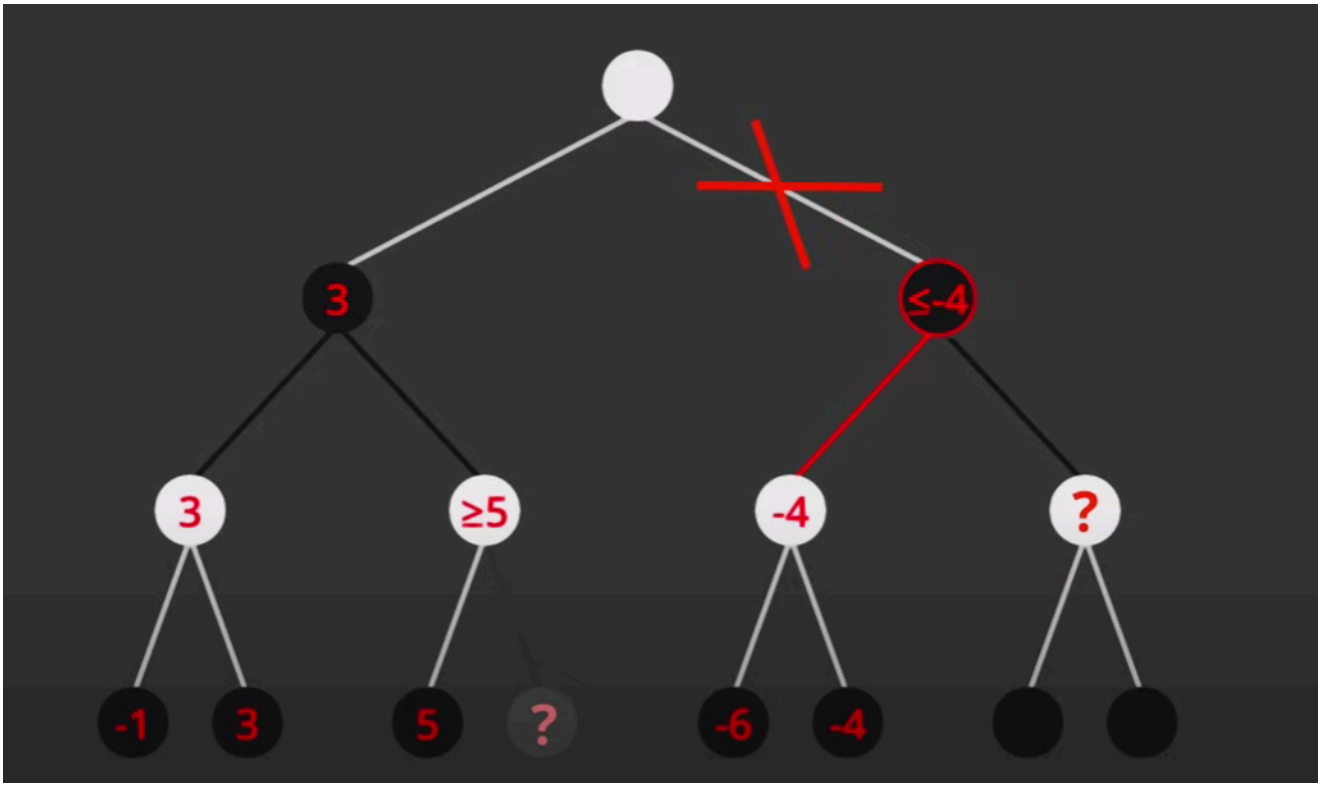
The two-parameter can be defined as: Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞. Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

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: α>=β 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.



In the above position we can see that the left side of the tree is pruned because black wants to minimize its value and 3 is smaller than 5 so it doesn't need to calculate the values of the other nodes and can cut off / prune the left side of the tree and take 3 as its value/evaluation.



Similarly here white does not require calculating the evaluation of the left side of tree since 3 is bigger than -4 (condition for pruning is satisfied).

Code for Pruning

```
function minimax(position, depth, alpha, beta, 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, alpha, beta, false)
         maxEval = max(maxEval, eval)
         alpha = max(alpha, eval)
         if beta <= alpha
            break
      return maxEval

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

Here we have added two more variables `#alpha` and `#beta` these will keep track of the highest and lowest evaluation respectively. To apply pruning we will set the highest possible evaluation b/w alpha and current evaluation and then comparing it with beta and if beta is less than or equal to alpha we break/prune and return the max evaluation.

Similarly for the minimizing player we will set the value of `#beta` to whatever is smaller b/w beta and latest evaluation and then compare its value with alpha and if beta is less than or equal to alpha we break the loop and return the minimum evaluation.