Lab Assignment No. 3

Aim:

Implement Alpha-beta Tree search for any game search problem[TIC TAC TOE Game]

Objective:

Students will be able to understand how Alpha-beta Tree search is used in any game search problem

Software and Hardware Requirement: Open Source python Programming tool /java and Ubuntu.

Theory:

- 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:

   a. **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 **-∞**.
   b. **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.
- The main condition which required for alpha-beta pruning is:

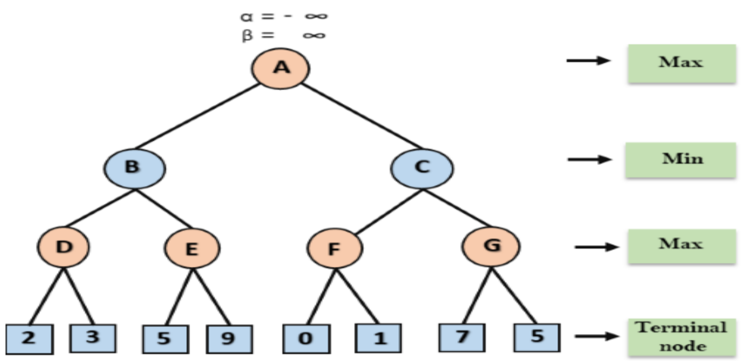
$\alpha >= \beta$

- 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:

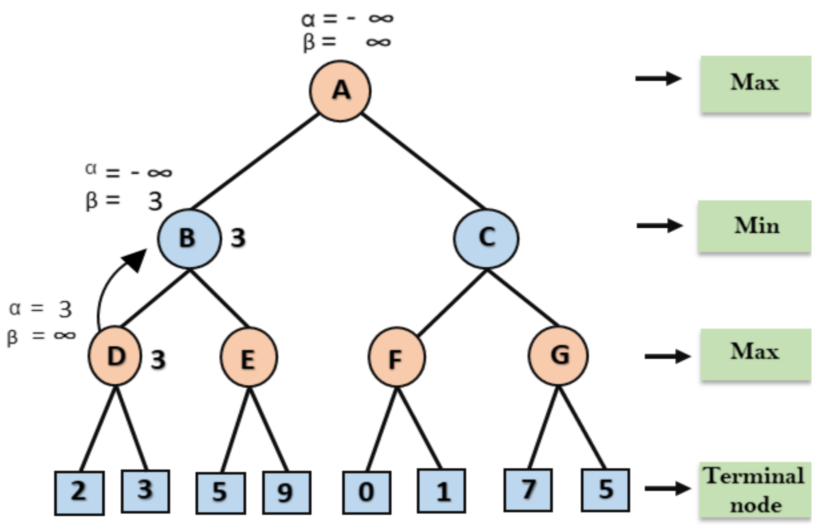
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

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.



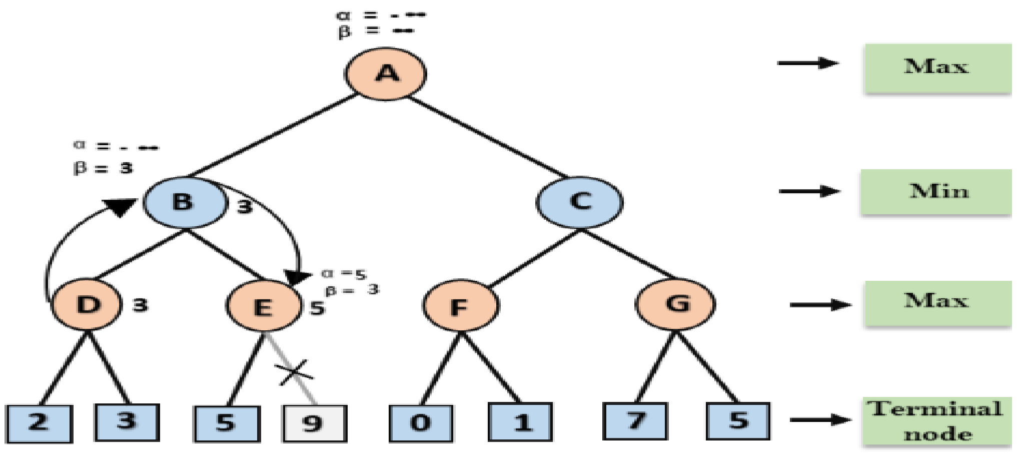
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

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.

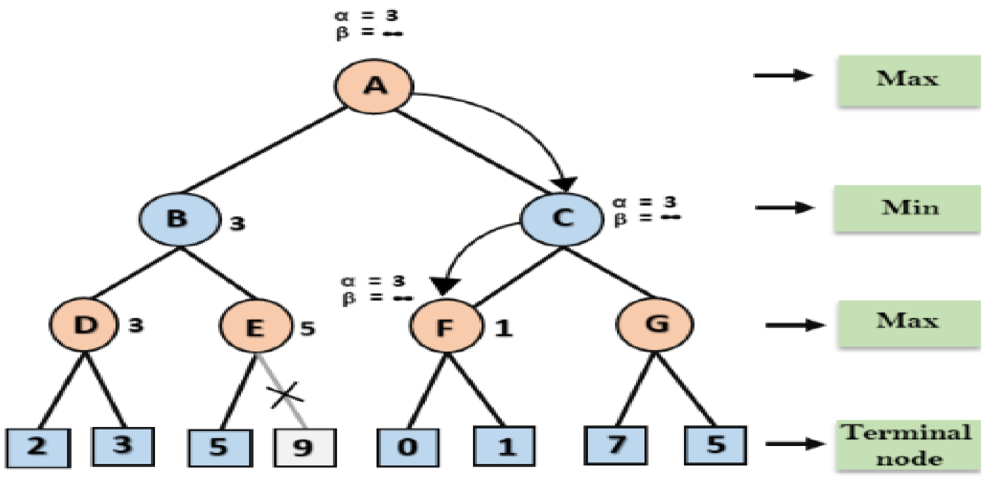


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max

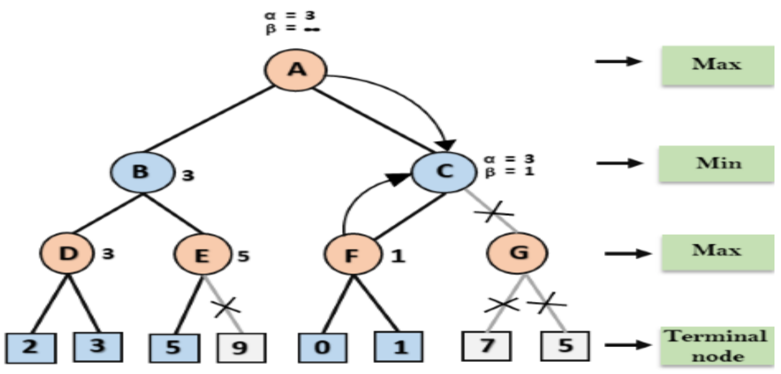
(-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

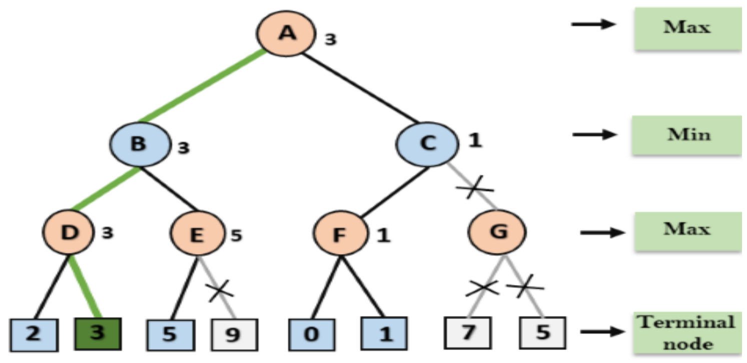
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



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.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

Algorithm:

1. First, generate the entire game tree starting with the current position of the game all the way up to the terminal states.
2. Apply the utility function to get the utility values for all the terminal states.
3. Determine the utilities of the higher nodes with the help of the utilities of the terminal nodes. For instance, in the diagram below, we have the utilities for the terminal states written in the squares.
4. Calculate the utility values with the help of leaves considering one layer at a time until the root of the tree.
5. Eventually, all the backed-up values reach to the root of the tree, i.e., the topmost point. At that point, MAX has to choose the highest value.

1. **Algorithm for a Tic-Tac-Toe game** that allows a player to play against a computer AI.
2. The game board is represented as a 3x3 grid, where empty cells are denoted by 0, 'X' is represented by 1, and 'O' is represented by -1.
3. There are functions to display the game board (`Gameboard`), clear the board (`Clearboard`), check for a winning player (`winningPlayer`), check if the game has been won (`gameWon`), print the result of the game (`printResult`), find blank cells on the board (`blanks`), check if the board is full (`boardFull`), and set a move on the board (`setMove`).
4. The `playerMove` function allows a human player to make a move by entering a number between 1 and 9, corresponding to the position on the board.
5. There are AI players, 'O' (`o_comp`) and 'X' (`x_comp`), which use the minimax algorithm with alpha-beta pruning to make optimal moves. Initially, they make random moves for the first turn, and then they calculate the best moves using minimax for subsequent turns.
6. The `makeMove` function is used to make a move for a player (human or AI) based on the mode (1 for player vs. AI, 2 for AI vs. AI).
7. The `pvc` function allows a player to choose whether they want to play first or second against the computer. Then, it initializes the game and alternates turns between the player and the computer until there is a winner or a draw.
8. Finally, the game is started by calling the `pvc()` function.

Overall, this code creates a functioning Tic-Tac-Toe game where you can play against an AI opponent, and the AI uses a minimax algorithm to make smart moves.

Applications:

Alpha Beta Pruning is an optimization technique of the Minimax algorithm. This algorithm solves the limitation of exponential time and space complexity in the case of the Minimax algorithm by pruning redundant branches of a game tree using its parameters Alpha(α) and Beta(β).

Input: `Enter a number between 1-9`

Output:

```
Enter a number between 1-9: 3
| O || O || X |
---------------
| X || X || O |
---------------
| O ||   || X |
---------------
===============
| O || O || X |
---------------
| X || X || O |
---------------
| O || O || X |
---------------
===============
Draw
```

Conclusion:

Thus we have implemented Tic Tac Toe game using Alpha-beta Tree search algorithm.

Outcome:

Upon completion of this experiment, students will be able to:

Implementation of alpha beta tree search for Tic Tac Toe game search problem.

Questions:

1. What is the Minimax algorithm?

2. Why is the time complexity of the Minimax algorithm so high?

3. What are the best case and worst case time complexity of alpha-beta pruning?

4. Can alpha-beta pruning be used for applications apart from games?