Name: Vansh Yelekar

Roll No: 332066 TY_B3

PRN: 22210183

Assignment: 04

Title:

Implementation of Min-Max Search Procedure with alpha beta pruning for finding the solutions of games.

Theory:

• Min-Max Algorithm:

The Min-Max search is a recursive algorithm used in two-player games like chess and tic-tac-toe to find the optimal move. The algorithm works on the principle of minimizing the loss probability for the worst-case scenario. It assumes both players are playing well. The algorithm recursively evaluates all possible moves, constructing a game tree where:

o Max nodes represent the current player's move, aiming to maximize their advantage.
o Min nodes represent the opponent's move, aiming to minimize the current player's advantage.

• Alpha-Beta Pruning:

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm. It reduces the number of nodes evaluated in the game tree by eliminating branches that cannot influence the final decision. This is achieved by maintaining two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively.

- Alpha: The best (highest) value that the maximiser can guarantee given the current state.
- Beta: The best (lowest) value that the minimizer can guarantee given the current state.

As the algorithm traverses the tree, it updates these values. If it finds a move that is worse than the current alpha for the maximiser or beta for the minimizer, it prunes (cuts off) that branch, as it cannot affect the outcome.

During the search:

- Alpha is updated for maximiser nodes. o Beta is updated for minimizer nodes.

- When a node's alpha is greater than or equal to beta, further exploration of that branch is unnecessary (we "prune" it).

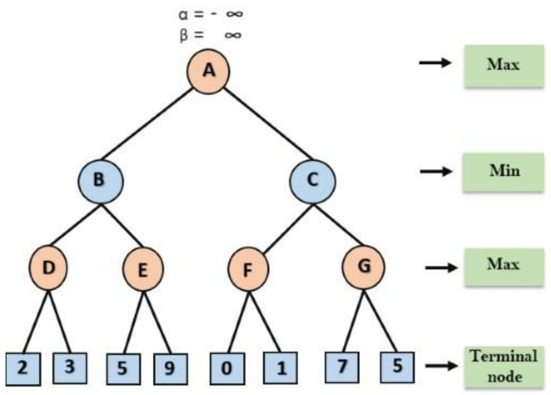
Key points about alpha-beta pruning:

- 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.

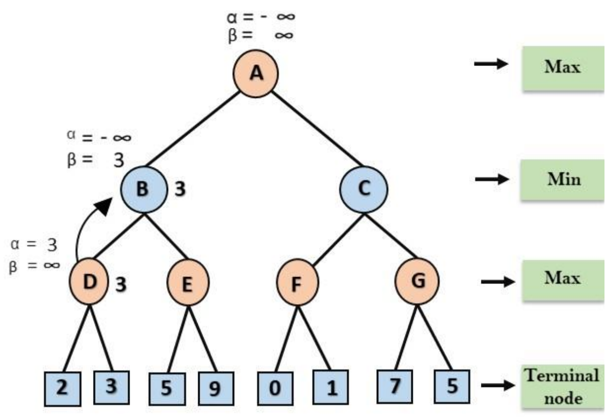
Working Of Alpha-Beta Pruning:

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.

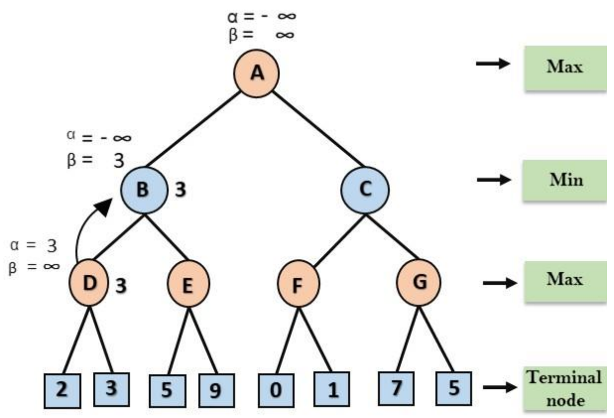


- Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.
- Step 3: Now algorithm backtrack to node B, where the value of β will change as this is 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.



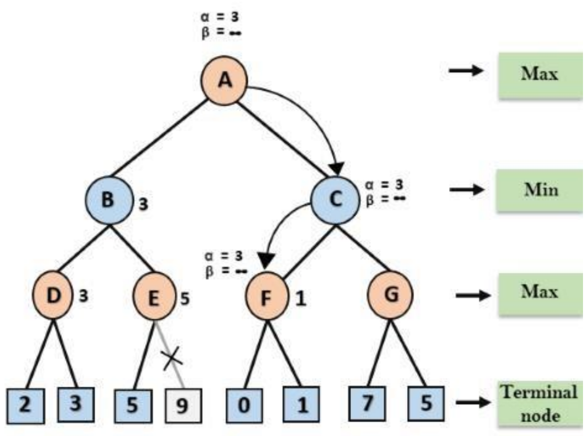
In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

- 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.

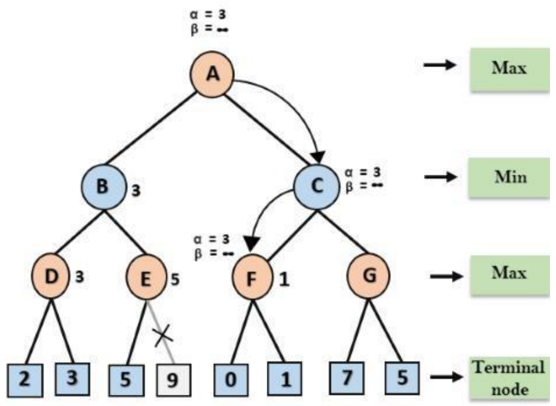


- Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
  At node C, α=3 and β= +∞, and the same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

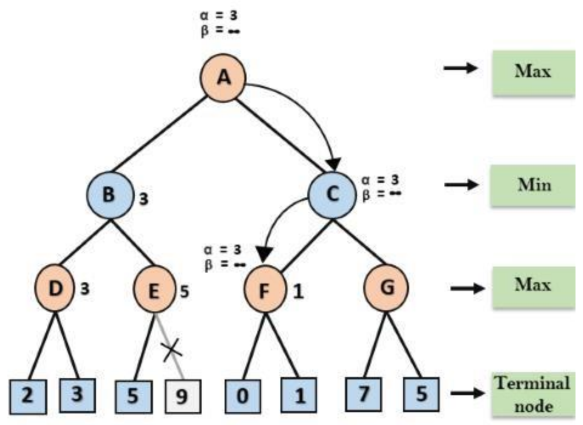
6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



- 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.



o 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.



Implementation:

General Template:

```
#include <iostream>

#include <vector>

#include <limits.h>  // For INT_MAX and INT_MIN


using namespace std;


// Function to evaluate the static score of a node (game state)

int evaluateNode(int node) {

   // Implement the static evaluation function based on the game

   // Placeholder: returning the node itself for simplicity

return node;

}


// Function to check if the node is a terminal node (game over or max depth

reached) bool isTerminalNode(int depth, int node) {

   // Placeholder for actual terminal node check

   // In real use, check for game over (win, loss, draw) or depth limit

return depth == 0;

}
```

```
// Function to generate children of a node (all possible moves from current state)
vector<int> getChildren(int node) {
   // Placeholder for generating children of the node
   // In real use, this would return valid game states from the current state     return
{node - 1, node + 1};  // Example: just decreasing or increasing the node value
}


// Minimax function with Alpha-Beta Pruning
int minimax(int node, int depth, int alpha, int beta, bool maximizingPlayer) {
   // Base case: terminal node or maximum depth
reached     if (isTerminalNode(depth, node)) {         return
evaluateNode(node);
   }


   if (maximizingPlayer) {  // Maximizer's turn         int
maxEva = INT_MIN;  // Initialize to negative infinity


      // Generate and explore all children
for (int child : getChildren(node)) {
         int eva = minimax(child, depth - 1, alpha, beta, false);
         maxEva = max(maxEva, eva);  // Maximizer chooses the maximum value
alpha = max(alpha, maxEva); // Update alpha


         //           Alpha-Beta
pruning              if (beta <=
alpha) {                break;  //
Cut-off
         }
      }
      return maxEva;     }
else {  // Minimizer's turn
      int minEva = INT_MAX;  // Initialize to positive infinity
```

```
        // Generate and explore all children
    for (int child : getChildren(node)) {
            int eva = minimax(child, depth - 1, alpha, beta, true);             minEva
    = min(minEva, eva);  // Minimizer chooses the minimum value             beta
    = min(beta, minEva);   // Update beta

            //            Alpha-Beta
    pruning              if (beta <=
    alpha) {               break;  //
    Cut-off
            }
        }
        return minEva;
      }
    }
    int main() {
      int node = 10;           // Example node (game state)
    int depth = 5;           // Maximum depth for the
    search    int alpha = INT_MIN;      // Initial value for
    alpha     int beta = INT_MAX;       // Initial value for
    beta
      bool maximizingPlayer = true;  // Assume the first move is by the Maximizer

      // Call minimax and print the result
      int bestMove = minimax(node, depth, alpha, beta, maximizingPlayer);
    cout << "Best move evaluation: " << bestMove << endl;

      return 0;
    }
```

OUTPUT:

```
        Best move evaluation: 11
```

➢ **Conclusion:**

The Minimax set of rules with Alpha-Beta pruning is a powerful approach for solving two-participant, zero-sum games with ideal data, inclusive of Chess, Tic-Tac-Toe, Checkers, and Connect Four. Minimax explores all viable actions and counter-movements to decide the premier approach for both gamers. Alpha-Beta pruning complements its efficiency by using cutting off branches in the sport tree that don't have an effect on the very last selection, permitting deeper exploration whilst keeping the equal end result.

While effective for deterministic video games, it will become impractical for video games with hidden information (e.g., Poker) or randomness (e.g., Backgammon). Alpha-Beta pruning makes the algorithm greater scalable for complex video games but continues to be computationally disturbing for actualtime programs, regularly requiring heuristic functions for deeper searches. Despite these boundaries, Minimax with Alpha-Beta pruning stays fundamental in AI recreation techniques.