Game Tree Search Algorithm

We consider only two persons perfect information games such as chess, tic-tac-toe etc. played by two players who move in turn. They respectively try to maximize or minimize a scoring function also called utility function. They each know completely what both the players have done and can do. i.e, all information about the board position and moves are available to the person. There is no chance factor like ~~throw~~ throwing dice games.

Game playing is an intellectual activity. Our attempt is to achieve the ~~art~~ intellectual ability of the computers which inturn will increase the intellectual ability of the humans.

Game Tree →

↳ The search used is adversarial search)

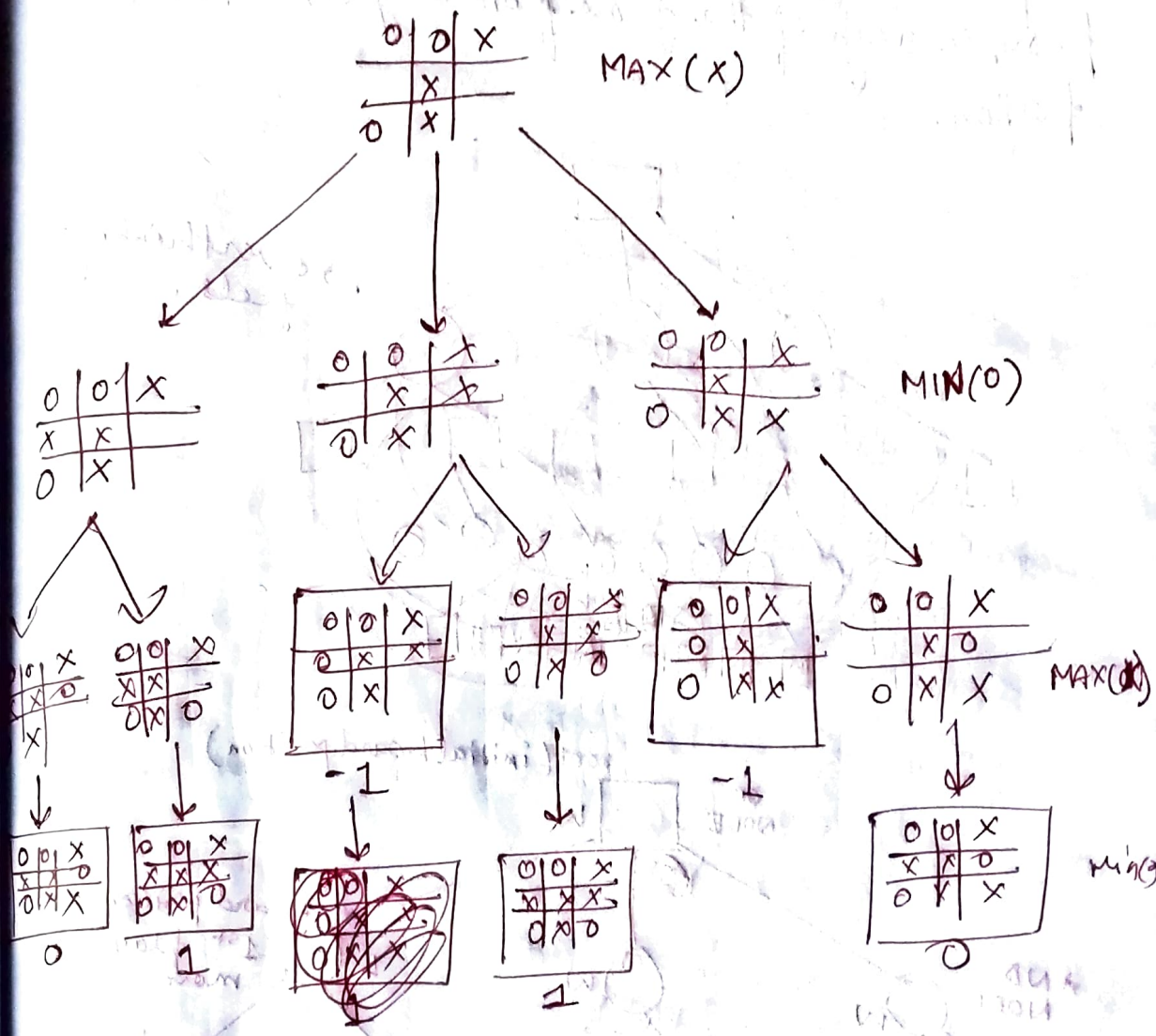
→ Adversarial Search can be represented as a tree where the nodes represent the current state of the game and the arcs represents the moves. The game tree consists of all possible moves for the current player, starting at the root and all possible moves for the next player as the children of these nodes and so forth as far to the future of the game as desired.

Each individual move by one player is called a ply.

The leaves of a game tree represent terminal positions where the outcome of the game is clear (a win, a loss, a draw) Each terminal position has a score. Highest scores are good for one of the player

called the max player. The other player called the min player tries to minimize the score.

For e.g. → We associate +1 with a win, 0 with a draw and -1 with a loss for max player in the game of tic-tac-toe.



⇒ Formulation of the game:

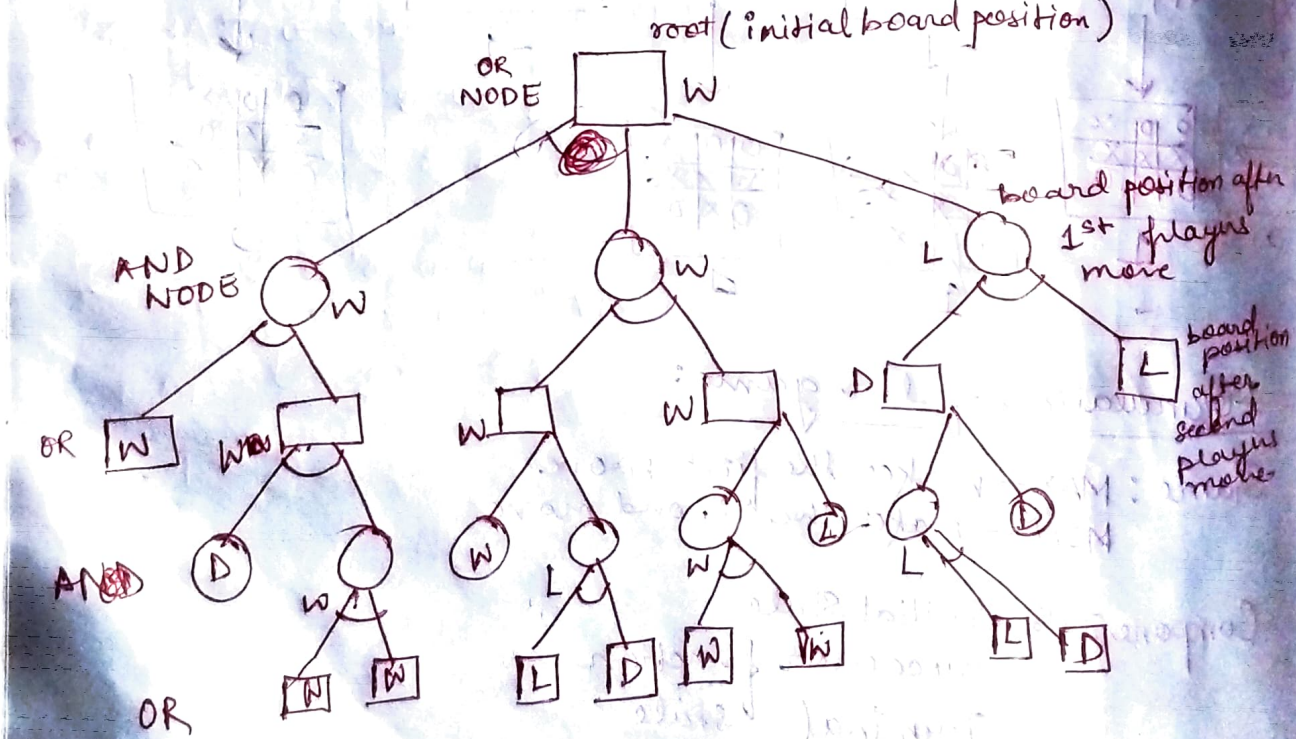
Game : MAX – makes the first move
MIN – makes the second move

Components: Initial State
Successor function
Terminal state
Utility function – give numerical value to all the terminal state.
High value means good for MAX and bad for MIN and vice-versa.

A game begin from an initial state and ends in a position that.
using a simple criteria can be declared as a win or loss or draw.

A game tree is an explicit representation of all possible ways of the game from a given board position.

→ overthinking feels.



W ≡ win
L ≡ Loss
D ≡ draw

Each path from a root to a terminal node represents a different complete play of a game.
(Complete refers to get to a conclusion)

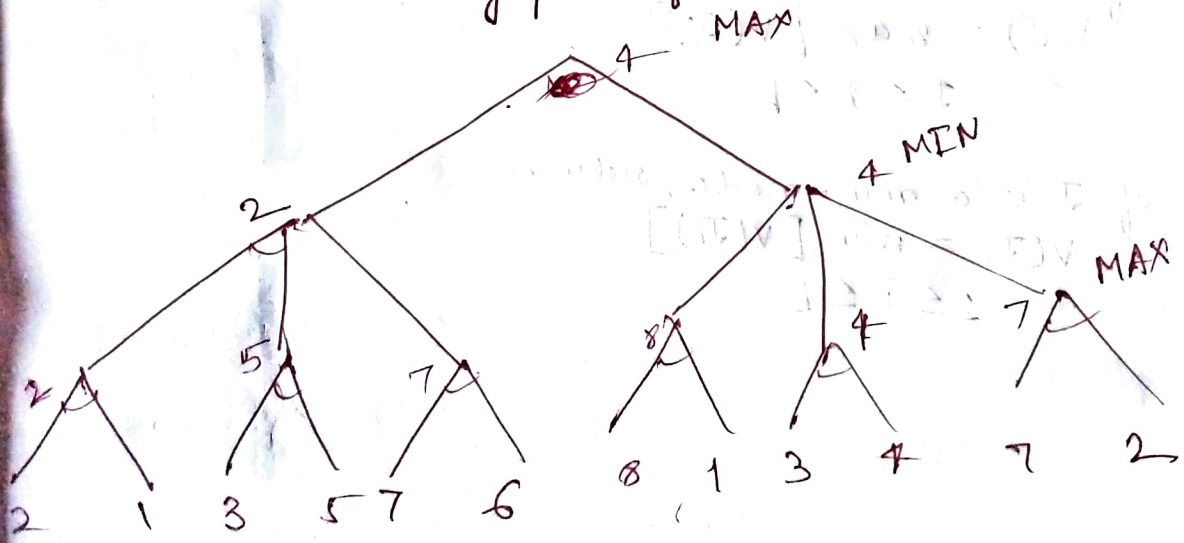
Given a game tree it is easy to find a winning play for a player. A tree is nothing but a AND-OR tree since

(a) For a game position J playing the game means - to find a winning strategy from J to computer C.

(b) If C is to move the problem is solved if winning strategy can be found by any of its successors, hence OR-node.

(c) If H is to move the problem is solved with respect to C if C can force a win from each of H's node hence AND node

Since easy to write programs to win as execution evaluation of all plays will guide the computer to find its best move at any point of time



→ MINI-MAX Rule

(1) The value V(J) of a node J of the search frontier is its static evaluation score ~~etc~~. Otherwise,

(2) If J is the max node, V(J) is the maxm value of any of its successors

(3) If J is the ~~mean~~ min node, V(J) is the minm value of any of its successors.

Algorithm MINI-MAX →

To determine V(J), do the following.

(1) If J is a terminal value return V(J) = B(J) otherwise

(2) generate J's successors say J_1 J_2 ... J_b where b is the branching factor.

(3) Evaluate V(J1), V(J2) ... V(Jb) from left to right

(4) If J is a max-node, return

$$V(J) = \max_{1 \le i \le b} [V(J_i)]$$

(5) If J is a min-node, return

$$V(J) = \min_{1 \le i \le b} [V(J_i)]$$

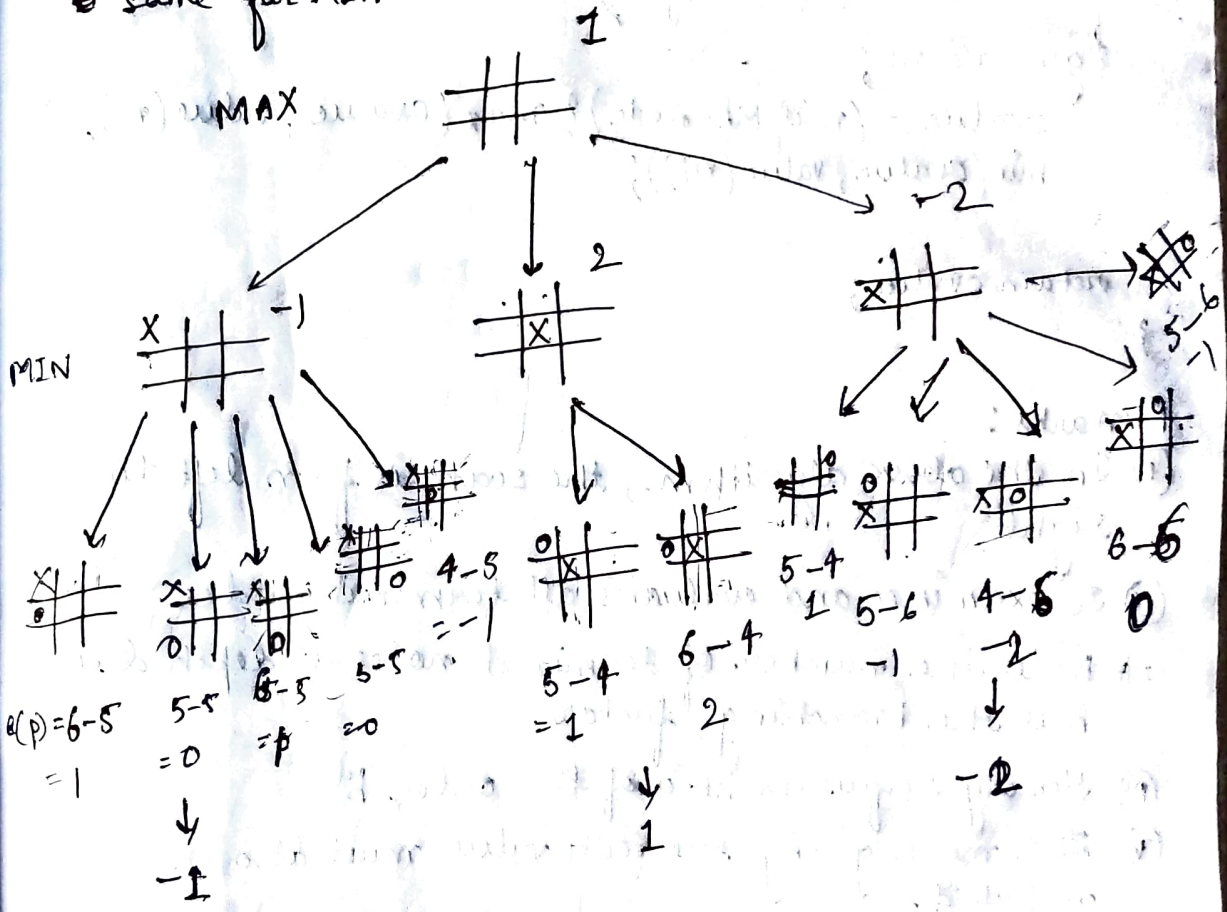
5/07/22

#Example →

Let's assume that MAX (the computer) marks (X) and MIN marks circles (O) and marks in the game of tic-tac-toe.

Let the static evaluation score e(p) = (no. of complete rows, columns, diagonals that are still "OPEN" for MAX)

- same for MIN



Disadvantage:

→ To use this method, we have to

(i) generate the full game tree first

(ii) Evaluate all the terminal nodes

(iii) Examine the full game tree

Recursive Version of MINI-MAX Algorithm

```
main()
{
    V = value(root);
    output V;
}
value(node n)
{ int cvalue; /* cvalue is current value */
  if (n is a terminal) return e(n); /* defined by the
  cvalue = (n is MIN node) ? ∞ :- ∞          depth bound */
  for each successor ni of n)
  { generate ni;
    cvalue = (n is MAX node) ? max(cvalue, value(ni)) :
    min(cvalue, value(ni));
  }
  return cvalue;
}
```

Remarks:

(i) In the above algorithm, the scan is from left to right

(ii) It examines and evaluates all terminal nodes

(iii) b^d is the number of terminal nodes at depth d if b is the branching factor

(iv) Memory requirement is of the order b^d

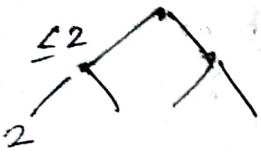
(v) The move giving the root value must also be outputted.

(vi) The game tree search algorithm attempts to find the value of the root by examining minimum no. of terminal node (also ~~cutting~~ intermediate nodes). The no. of terminal leaf nodes examined serves as the measure of performance of the algorithm since calculation of state evaluation score of a terminal leaf is most time consuming and costly.

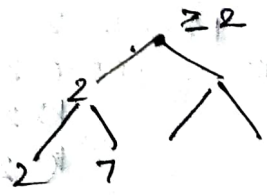
α-β pruning

↳ It is a backtracking strategy.

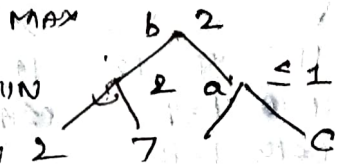
↳ This is to reduce the no. of terminal nodes examined. By the MINI-MAX procedure, we evaluate all four terminal nodes but we need not even look at the last node C, since at A, the score is less than or equal to 1 and at B, the score is greater than or equal to 2. Thus at B, the score must be 2 irrespective of the quality number at C.



(I)

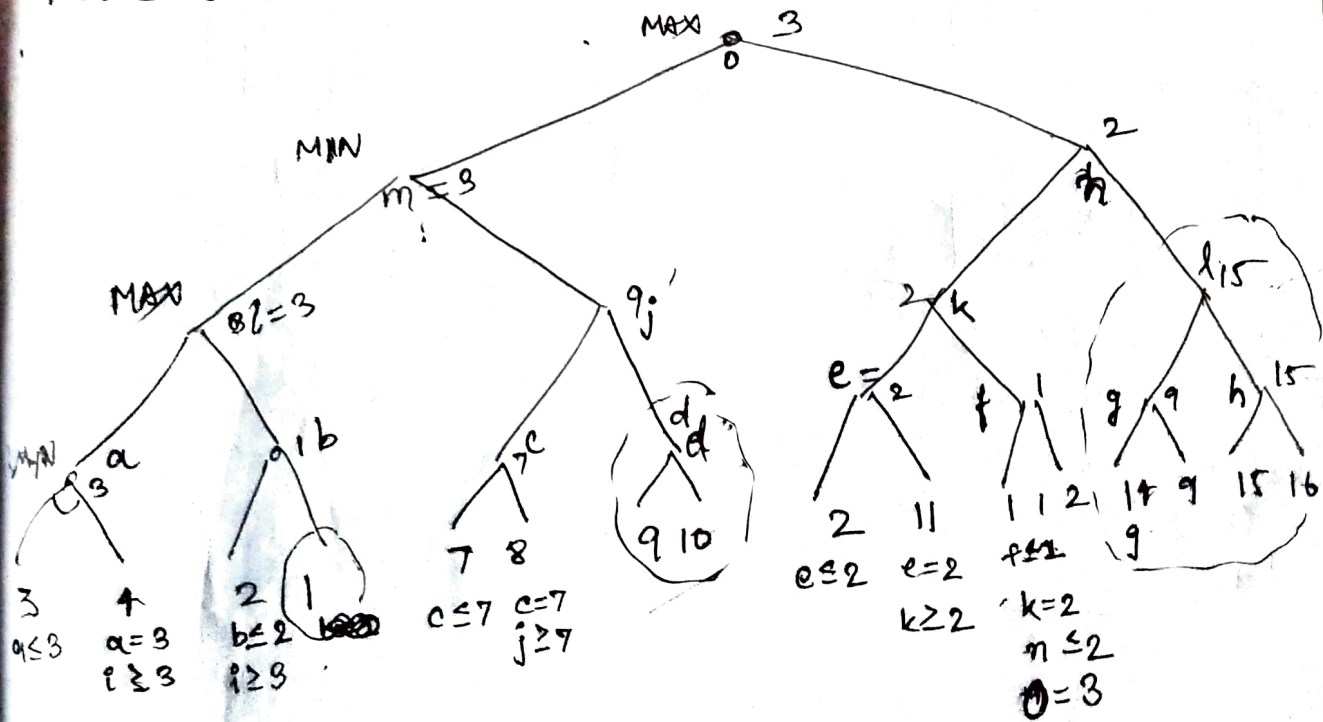


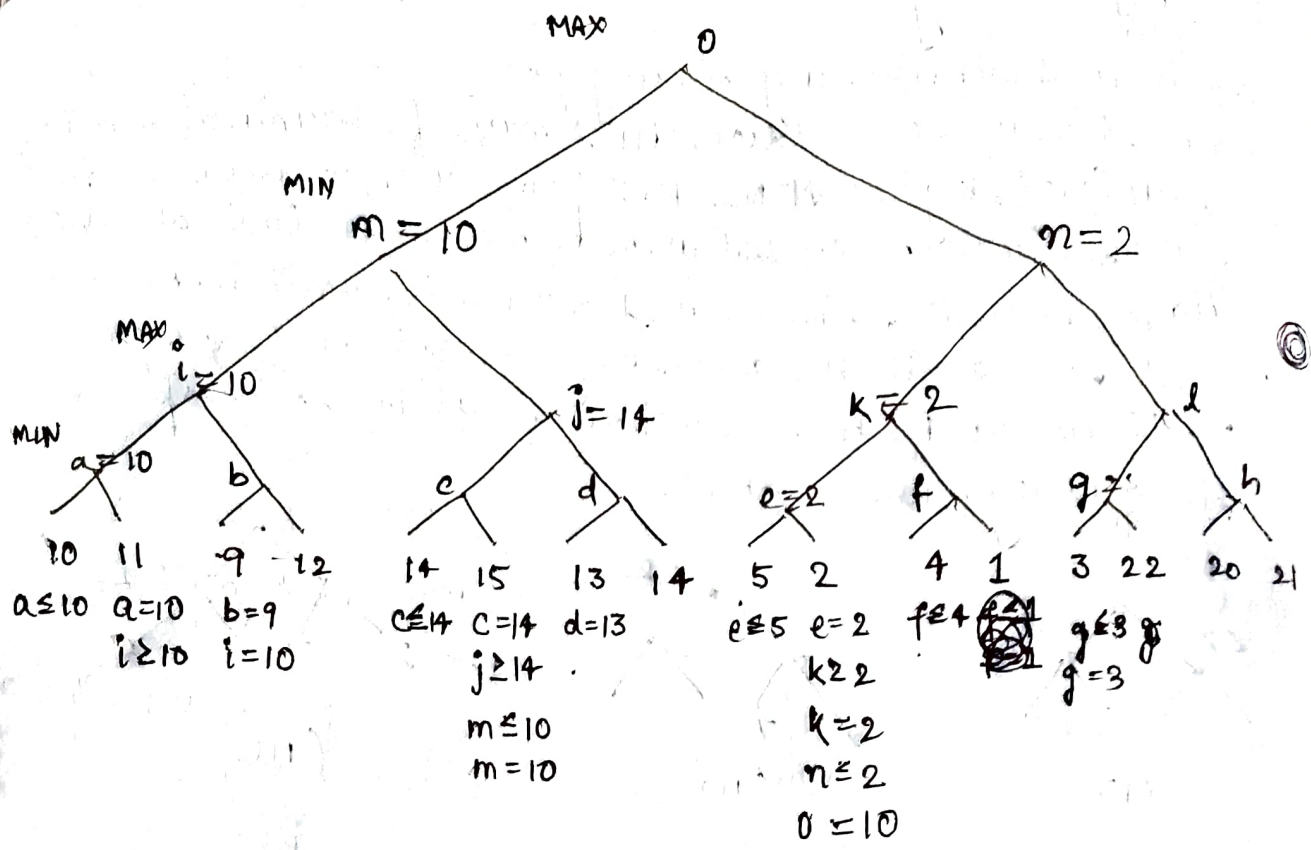
(II)



(III)

Max node value → α Min node value → β




MAX
0
MIN
m = 10
n = 2
MAX
i ≥ 10
j = 14
k ≤ 2
l
MIN
a = 10
b
c
d
e ≤ 2
f
g
h
10 11 9 12
14 15 13 14
5 2 4 1 3 22 20 21
a ≤ 10 a = 10 b = 9
i ≥ 10 i = 10
c ≤ 14 c = 14 d = 13
j ≥ 14
m ≤ 10
m = 10
e ≤ 5 e = 2
k ≥ 2
k = 2
n ≤ 2
0 = 10
f ≤ 4
g ≤ 3
g = 3

#

Let the game tree have a branching factor b and depth d, then the minimum possible number of static evaluations required to solve a game tree is

$$E = b^{\lfloor d/2 \rfloor} + b^{\lceil d/2 \rceil} - 1$$

$$= \begin{cases} 2b^{\lceil d/2 \rceil} - 1 & \text{if } d \text{ is even} \\ b^{\frac{d-1}{2}} + b^{\frac{d+1}{2}} - 1 & \text{if } d \text{ is odd.} \end{cases}$$

⇒ From the diagram, $d = 4$, $b = 2$

$$E = 2^{\frac{4}{2}} + 2^2 - 1$$

$$= 7$$

$$= 2^{3/2} + 2^{5/2} - 1$$

$$= (\sqrt{2})^3 + (\sqrt{2})^5 - 1$$

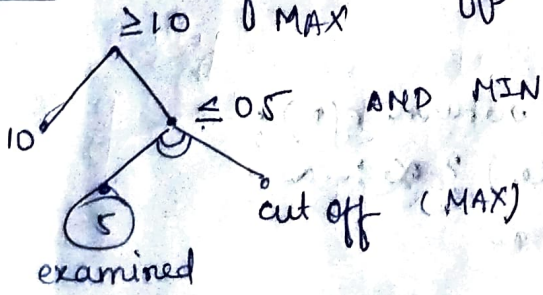
$$= 2\sqrt{2} + 4\sqrt{2} - 1$$

Remarks: We can say the bound on a node as a current value of a node. Actual value of a node is said to its current value

(i) If all the sons are yet to be examined

(ii) If sum of the sons are yet to be examined but their examinations can in no way affect the value of the root.

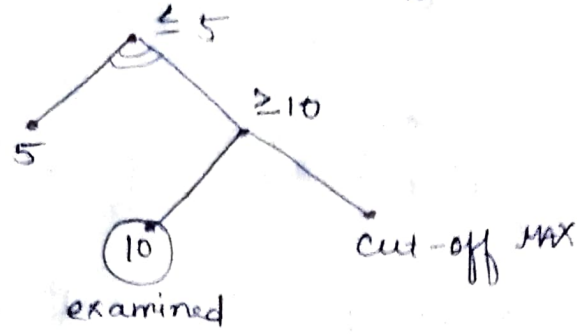
Case I: (Case of α - cut off)



The MAX ancestor of a MIN node cuts-off the sons of a MIN node, if the current value of the MIN-node is ≤ the current value of the MAX ancestor.

(Reason – Since current value of the MIN node in such a ... can never increase the value of the MAX – ancestor

Case II : (Case of β – cutoff)



The MIN ancestor of a MAX-node cuts off the ... of the MAX-node if the current value of the MAX node is ≥ the current value of the MAX ancestor.

(Reason – Since current value of the max-node in such a case can never decrease the value of the MIN ancestor.

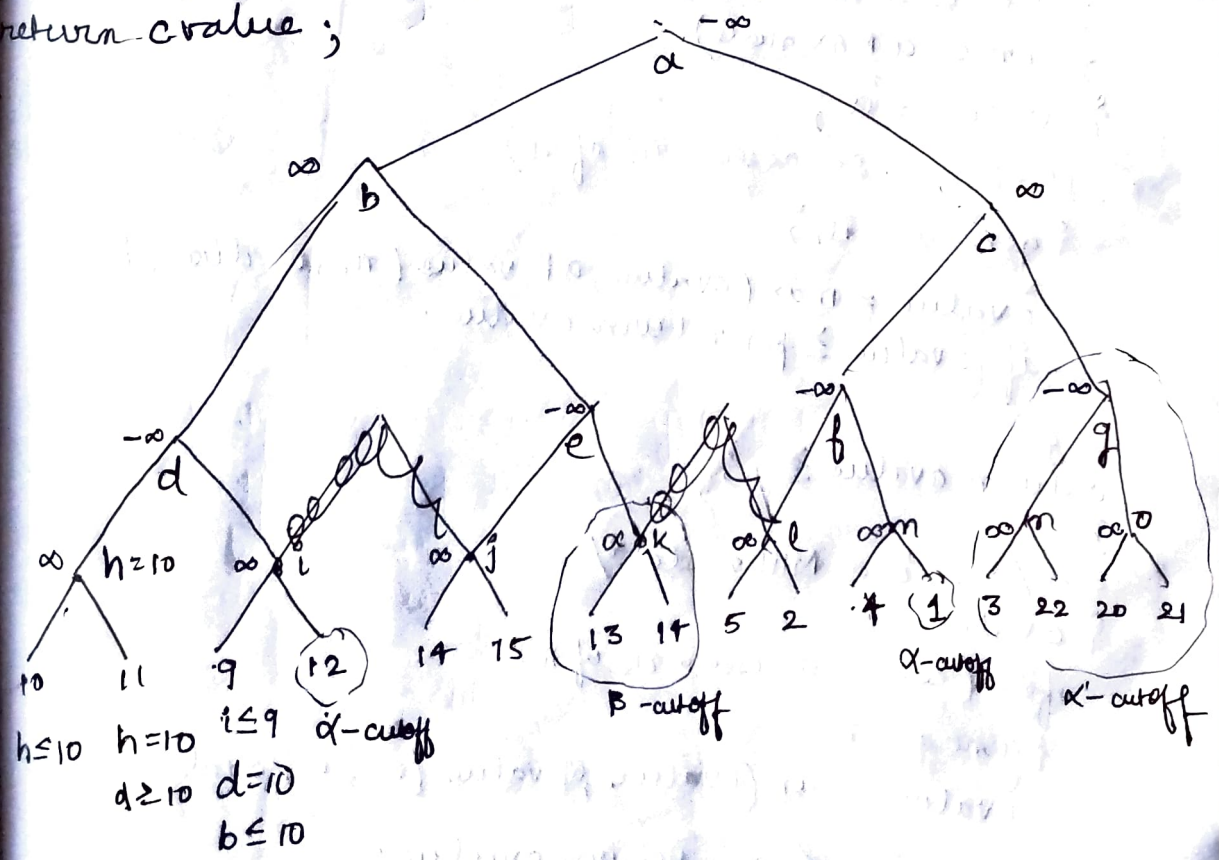
The current value of the MAX node → α – value or α – bound

The current value of the MIN node → β – value or β – bound

Modification of MINI – MAX Algorithms :

```
main ()
{ v = value (root);
    output v;
}
value (node n)
{ int evalue;
  if (n is a terminal node) return e(n);
  cvalue = (n is a MIN node) ? ∞ : -∞;
  for each successor ni of n
  { generate ni;
  if (n is a MAX node)
```

{ cvalue = max (cvalue, value(ni));
If (cvalue > minimum of all MIN ancestor of n)
{ cut off all son of n;
break;
}
}
else /* n is a MAX node */
{ cvalue = min (cvalue, value(ni));
If (cvalue < maximum of all MAX ancestors)
{ cutoff all sons of n;
break;
}
}
return cvalue;



α- Bound

→ It is a cut-off bound for a MIN-node J and is equal to the highest current value of all MAX ancestors of J i.e., highest current lower bound of MAX-ancestors of J. When the current value of the MIN node J is less than or equal to

α-bound. For if the sons of node j are pruned and cut-off takes place.

β-bound.

→ change MIN → MAX

α-β pruning algorithm →

```
main()
{ V = αβvalue (root, -∞, ∞)  /* α = -∞, β = ∞ */
  output V;
}
αβvalue (node n, int α, int β)
{ int cvalue;
  if (n is a terminal) return e(n);
  if (n is a MAX node)
  { cvalue = α;
    for (each successor ni of n)
    { generate ni;
      cvalue = max (cvalue, αβ value (ni, cvalue, β));
      if (cvalue ≥ β) return cvalue;
    }
    return cvalue;
  }
  else /* n is a MIN node */
  { cvalue = β;
    for (each successor ni of n)
    { generate ni
      cvalue = min (cvalue, αβvalue (ni, α, cvalue));
      if (cvalue ≤ α) return cvalue;
    }
    return cvalue;
  }
}
```

Worst case situation
Highest value successors first for MIN nodes.
Lowest value successors first for MAX nodes.