Informed Search Algorithms-II

(A*Algorithm)

A* Algorithm- Introduction

- A* (pronounced as A- star search) is the most widely and improved form of the best-first search.
- It evaluates each node n using a function f(n) which is combination of g(n) and h(n).

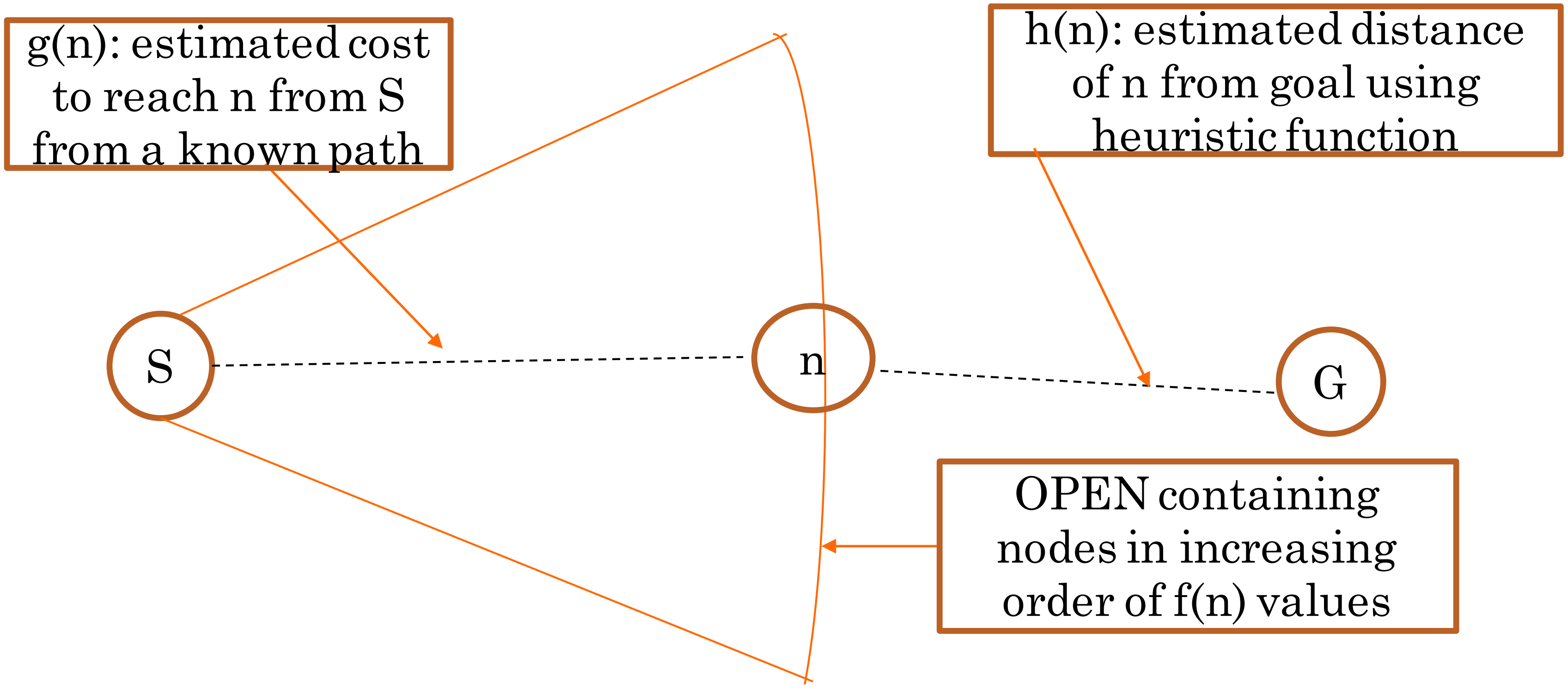
$$f(n) = g(n) + h(n)$$

where g(n) is the actual cost to reach the node n from starting state through some known path.

h(n) is the estimated distance of the node n from goal node (computed using heuristic function).

- The algorithm also maintains OPEN as a priority queue where the nodes are prioritized according to their f-values.
- Each node in OPEN is maintained as (node, parent, f(node))

A* ALGORITHM- INTRODUCTION CONTD…..


g(n): estimated cost to reach n from S from a known path
h(n): estimated distance of n from goal using heuristic function
S
n
G
OPEN containing nodes in increasing order of f(n) values

- h(n) is defined by the heuristic function and **would never change for any node n**. Thus h(n) is the property of the node.
- g(n) is the estimated cost to reach n from S from a known path. As more and more paths are learnt, this cost will keep on changing. Thus **g(n) changes while the A* algorithm explores the search space. In general g(n) ≥ g*(n) where g*(n) denotes optimized cost to reach n from S.**

Optimality of A* Algorithm

- The necessary conditions for A* algorithm to be optimal is:
  1) The branching factor of the search space should be finite.
  2) The cost to move from one node to another should be greater than or equal to zero i.e. cost(i,j) ≥ 0 for all i, j.
  3) The heuristic is an **admissible heuristic. An admissible heuristic h(n) is a heuristic that always *underestimates* the cost to reach the goal i.e. h(n) ≤ h*(n) where h*(n) is the optimized cost to reach goal from n.**

  In other words, for every node n, the cost to reach the goal provided by the heuristic function is never greater than the optimized cost to reach the goal.
- If the above mentioned conditions are not satisfied then the algorithm is A algorithm but not A* (* for any search algorithm denotes optimal algorithm).

A* Algorithm Pseudocode

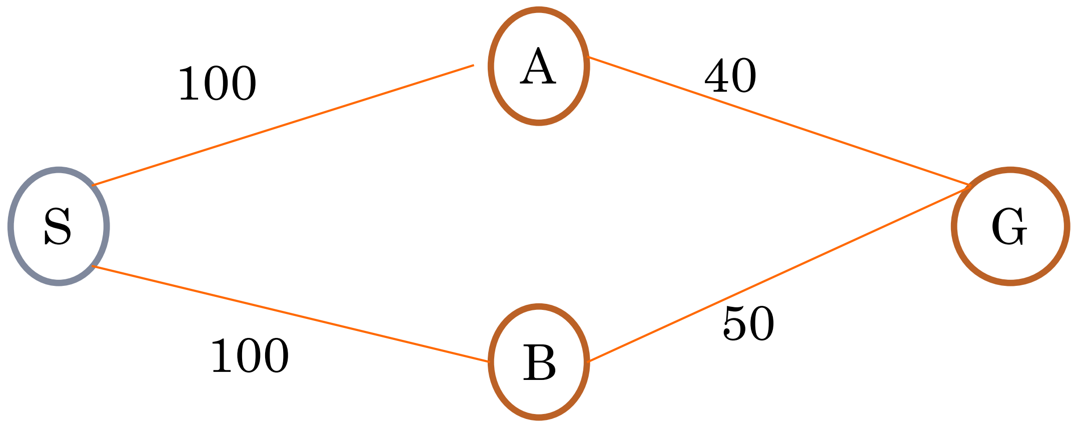
```
Insert start node (S,$\phi$,f(S)) to OPEN
Loop until OPEN is empty or success is returned
        (n,p,f(n)) $\leftarrow$ remove-head(OPEN) and add it CLOSED
        If n is a goal node return success and path to n.
        else
            successors $\leftarrow$ MOVEGEN(n)
                for each successor m do
                switch case
                    case 1: if m $\notin$ OPEN and m $\notin$ CLOSED
                            compute g(m) = g(n) + cost (n,m) and h(m)
                            compute f(m)=g(m) + h(m)
                            add (m,n,f(m)) to OPEN according to priority of f(m)
                    case 2: if m $\in$ OPEN
                            compute newg(m)=g(n)+cost(n,m)
                            if newg(m)<g(m) then
                                    update g(m) =newg(m) and f(m)=g(m)+h(m)
                                    update (m,n,f(m)) to OPEN according to priority of f(m)
                            end if
                    case 3: if m $\in$ CLOSED
                            compute newg(m)=g(n)+cost(n,m)
                            if newg(m)<g(m) then
                                    update g(m) = newg(m) and f(m)=g(m)+h(m)
                                    update (m,n,f(m)) to CLOSED according to priority of f(m)
                                    propogateimprovement(m)
                            end if
                end for
        end if
end loop
```

A* Algorithm Pseudocode Contd….

```
propagateimprovement(m)
for each successor s of m
        compute newg(s)=g(m) + cost(m,s)
        if newg(s) < g(s) then
                update g(s)=newg(s)
                update f(s)=g(s) + h(s)
                update (s,m,f(s)) in OPEN or CLOSED
                if s ∈ CLOSED then
                        propogateimprovement(s)
                end if
        end if
end for
```

Example I

- Consider the following search space with start node S and goal node G.



Consider following two heuristics defined for the problem:

- Heuristic 1: $h_1(S)=110$, $h_1(A)= 80$, $h_1(B) =70$
- Heuristic 2: $h_2(S)=110$, $h_2(A)= 30$, $h_2(B) =20$

Which of the two heuristic is admissible?

Using A* algorithm find the path from S to G using both heuristics and show that the path found using admissible heuristic is the optimal one?

Example 1 – Solution

Node	Cost to reach goal from different paths	Optimal Cost
S	140, 150	140 = h*(S)
A	40, 250	40 = h*(A)
B	50, 240	50 = h*(B)

As given,
Heuristic 1: $h_1(S)=110$, $h_1(A)= 80$, $h_1(B) =70$
Heuristic 2: $h_2(S)=110$, $h_2(A)= 30$, $h_2(B) =20$

For every node, heuristic 2 underestimates the optimal cost to reach goal whereas heuristic 1 is overestimating the optimal cost for nodes A and B.
Therefore, heuristic 2 is admissible and heuristic 1 is not

Example 1 – Solution Contd….

- **Path using heuristic 1**

$g(S)=0$, $h_1(S)=110$; $f(S)= 0+110$

Add (S,ϕ,0+110) to OPEN and CLOSED is empty

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }

Iteration I

Remove head node (S,ϕ,0+110) from OPEN and add to CLOSED.

Since S is not goal node, therefore successors of S, A and B are produced (case 1: both are new not in OPEN and CLOSED)

Successors of S:

A :$g(A)=g(S)+cost(S,A) = 0+100 =100$; $h_1(A) = 80$, $f(A) = 100+80$

B : $g(B)=g(S)+cost(S,B) = 0+100 =100$; $h_1(B) = 70$, $f(B) = 100+70$

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }
1	{(B,S,100+70)(A,S,100+80)}	{(S,ϕ,0+110)}

Example 1 – Solution Contd….

Iteration II

Remove head node (B,S,100+70) from OPEN and add to CLOSED.

Since B is not goal node, therefore successors of B i.e. S and G are produced (for S it is case 3 as it is already in CLOSED and for G it is case 1 which is not in OPEN and CLOSED)

Successors of B:

S :newg(S)=g(B)+cost(B,S) = 100+100 =200;

since newg(S) is not less than g(S), so this successor is ignored

G: g(G)=g(B)+cost(B,G) = 100+50=150; h_1(G) = 0 , f(G) = 150 + 0

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }
1	{(B,S,100+70)(A,S,100+80)}	{(S,ϕ,0+110)}
2	{(G,B,150+0)(A,S,100+80)}	{(S,ϕ,0+110) (B,S,100+70)}

EXAMPLE 1 – SOLUTION CONTD….

Iteration III

Remove head node (G,B,150+0) from OPEN and add to CLOSED.

Since G is goal node, therefore algorithm will stop and path $S \rightarrow B \rightarrow G$ is returned with path cost 150.

Example 1 – Solution Contd….

- **Path using heuristic 2**

$g(S)=0$, $h_2(S)=110$; $f(S)= 0+110$

Add (S,ϕ,0+110) to OPEN and CLOSED is empty

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }

Iteration I

Remove head node (S,ϕ,0+110) from OPEN and add to CLOSED.

Since S is not goal node, therefore successors of S, *i.e.* A and B are produced (case 1: both are new and not in OPEN and CLOSED)

Successors of S:

A :$g(A)=g(S)+cost(S,A) = 0+100 =100$; $h_2(A) = 30$, $f(A) = 100+30$

B : $g(B)=g(S)+cost(S,B) = 0+100 =100$; $h_2(B) = 20$, $f(B) = 100+20$

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }
1	{(B,S,100+20)(A,S,100+30)}	{(S,ϕ,0+110)}

Example 1 – Solution Contd….

Iteration II

Remove head node (B,S,100+20) from OPEN and add to CLOSED.

Since B is not goal node, therefore successors of B i.e. S and G are produced (for S it is case 3 as it is already in CLOSED and for G it is case 1 which is not in OPEN and CLOSED)

Successors of B:

S :newg(S)=g(B)+cost(B,S) = 100+100 =200;

since newg(S) is not less than g(S), so this successor is ignored

G : g(G)=g(B)+cost(B,G) = 100+50=150; h_2(G) = 0 , f(G) = 150 + 0

Add (G,B,150+0) to OPEN after (A,S,100+30)

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }
1	{(B,S,100+20)(A,S,100+30)}	{(S,ϕ,0+110)}
2	{(A,S,100+30) (G,B,150+0)}	{(S,ϕ,0+110) (B,S,100+20)}

Example 1 – Solution Contd….

Iteration III

Remove head node (A,S,100+30) from OPEN and add to CLOSED.

Since A is not goal node, therefore successors of A i.e. S and G are produced (for S it is case 3 as it is already in CLOSED and for G it is case 2 which is already in OPEN)

Successors of A:

S: newg(S) = g(A)+cost(A,S)= 100 + 100 = 200

since newg(S) is not less than g(S), so this successor is ignored

G : newg(G)=g(A)+cost(A,G) = 100+40=140; $h_2(G) = 0$, f(G) = 140 + 0

since newg(G) is less than g(G), therefore update (G,A,140+0) in OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+110)}	{ }
1	{(B,S,100+20)(A,S,100+30)}	{(S,ϕ,0+110)}
2	{(A,S,100+30) (G,B,150+0)}	{(S,ϕ,0+110) (B,S,100+20)}
3	{(G,A,140+0)}	{(S,ϕ,0+110) (B,S,100+20) (A,S,100+30) }

Example 1 – Solution Contd….

Iteration IV

Remove head node (G,A,140+0) from OPEN and add to CLOSED.

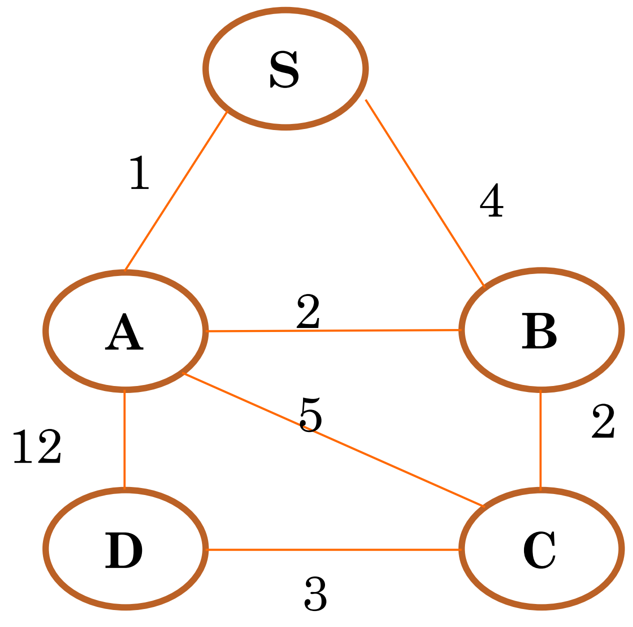
Since G is goal node, therefore algorithm will stop and path

S→A → G is returned with path cost 140.

Thus, the path returned by heuristic 2 (admissible heuristic) is optimal.

Example 2

- For the search space shown below, find the optimal path from S to D using the heuristic values defined in table.



Node	Heuristic Value
S	7
A	6
B	2
C	1
D	0

Example 2 – Solution

g(S)=0 , h(S)=7 ; f(S)= 0+7

Add (S,ϕ,0+7) to OPEN and CLOSED is empty

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+7)}	{ }

Iteration I

Remove head node (S,ϕ,0+7) from OPEN and add to CLOSED.

Since S is not goal node, therefore successors of S i.e. A and B are generated (case 1: both are new and not in OPEN and CLOSED)

Successors of S:

A :g(A)=g(S)+cost(S,A) = 0+1 =1; h(A) = 6 , f(A) = 1+6

B : g(B)=g(S)+cost(S,B) = 0+4 =4; h(B) = 2 , f(B) = 4+2

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+7)}	{ }
1	{(B,S,4+2) (A,S,1+6)}	{(S,ϕ,0+7)}

Example 2 – Solution Contd….

Iteration II

Remove head node (B,S,4+2) from OPEN and add to CLOSED.

Since B is not goal node, therefore successors of B i.e. S ,A and C are produced (for S it is case 3 as it is already in CLOSED, for A it is case 2 as it is already in OPEN and for C it is case 1 which is not in OPEN and CLOSED)

Successors of B:

S :newg(S)=g(B)+cost(B,S) = 4+4=8;

since newg(S) is not less than g(S), so this successor is ignored

A : newg(A)=g(B)+cost(B,A) = 4+2=6;

since newg(A) is not less than g(A), so this successor is ignored

C: g(C)=g(B) +cost(B,C) = 4 + 2 =6, h(C) =1 f(C)=6+1

Add (C,B,6+1) after (A,S,1+6) to OPEN [**node generated before has higher priority in case of same f values**]

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+7)}	{ }
1	{(B,S,4+2) (A,S,1+6)}	{(S,ϕ,0+7)}
2	{(A,S,1+6)(C,B,6+1)}	{(S,ϕ,0+7) (B,S,4+2)}

Example 2 – Solution Contd….

Iteration III

Remove head node (A,S,1+6) from OPEN and add to CLOSED.

Since A is not goal node, therefore successors of A i.e. S,B,D and C are produced (for S and B it is case 3 as it is already in CLOSED, for C it is case 2 which is already in OPEN, and for D it is case 1 as it is not in OPEN or CLOSED)

Successors of A:

S: newg(S) = g(A)+cost(A,S)= 1 + 1 = 2

since newg(S) is not less than g(S), so this successor is ignored

B : newg(B)=g(A)+cost(A,B) = 1+2=3; h(B) = 2 , f(G) = 3+2

since newg(B) is less than g(B), therefore update (B,A,3+2) in CLOSED and propagate improvement to Child C as (C,B,5+1) in OPEN

C: newg(C)= g(A) + cost (A,C) = 1+5 =6

since newg(C) is not less than g(C), so this successor is ignored

D: g(D) =g(A) +cost(A,D) = 1+12 =13, h(D)= 0, f(D)=13+0

ADD (D,A,13+0) after (C,B,5+1) to OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+7)}	{ }
1	{(B,S,4+2) (A,S,1+6)}	{(S,ϕ,0+7)}
2	{(A,S,1+6)(C,B,6+1)}	{(S,ϕ,0+7) (B,S,4+2)}
3	{(C,B,5+1) (D,A,13+0)}	{(S,ϕ,0+7) (B,A,3+2) (A,S,1+6) }

Example 2 – Solution Contd….

Iteration IV

Remove head node (C,B,5+1) from OPEN and add to CLOSED.

Since C is not goal node, therefore successors of C i.e. A ,B and D are produced (for A and B it is case 3 as it is already in CLOSED, for D it is case 2 which is already in OPEN)

Successors of C:

A: newg(A) = g(C)+cost(C,A)= 5 + 5 = 10

since newg(A) is not less than g(A), so this successor is ignored

B : newg(B)=g(C)+cost(C,B) = 5+2=7;

since newg(B) is not less than g(B), so this successor is ignored

D: g(D) =g(C) +cost(C,D) = 5+3 =8, h(D)= 0, f(D)=8+0

UPDATE (D,C,8+0) to OPEN

Iteration	OPEN	CLOSED
0	{(S,ϕ,0+7)}	{ }
1	{(B,S,4+2) (A,S,1+6)}	{(S,ϕ,0+7)}
2	{(A,S,1+6)(C,B,6+1)}	{(S,ϕ,0+7) (B,S,4+2)}
3	{(C,B,5+1) (D,A,13+0)}	{(S,ϕ,0+7) (B,A,3+2) (A,S,1+6) }
4	{(D,C,8+0)}	{(S,ϕ,0+7) (B,A,3+2) (A,S,1+6) (C,B,5+1)}

Example 2 – Solution Contd….

Iteration V

Remove head node (D,C,8+0) from OPEN and add to CLOSED.

Since D is goal node, therefore algorithm will stop and path

S→A → B → C → D is returned with path cost 8.

PERFORMANCE OF A*

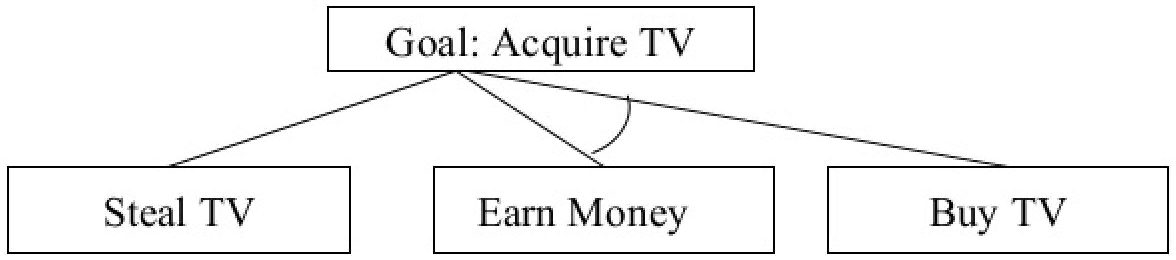
- **Completeness and Optimality:**
  - For finite, positive path costs, and admissible heuristics A* algorithm is always complete and optimal.

- **Space and Time Complexity**
  - Depends upon heuristic function.
  - For most of the problems, heuristics are never perfect. Therefore time and space usually grows exponentially.

AO* Algorithm

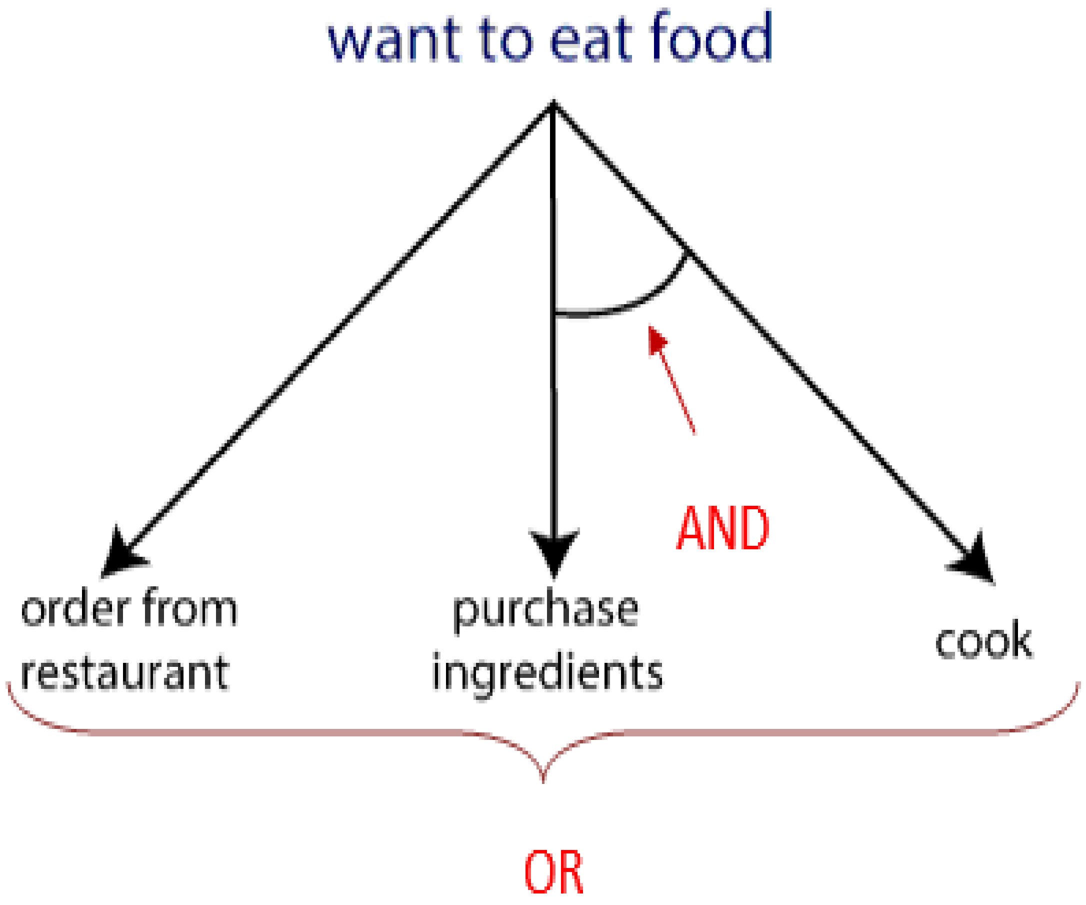
- **AND/OR graph**
- AND-OR graph is useful for representing the solution of problems that can be solved by decomposing them into a **set of smaller problems**, all of which must then be solved.

Problem Reduction

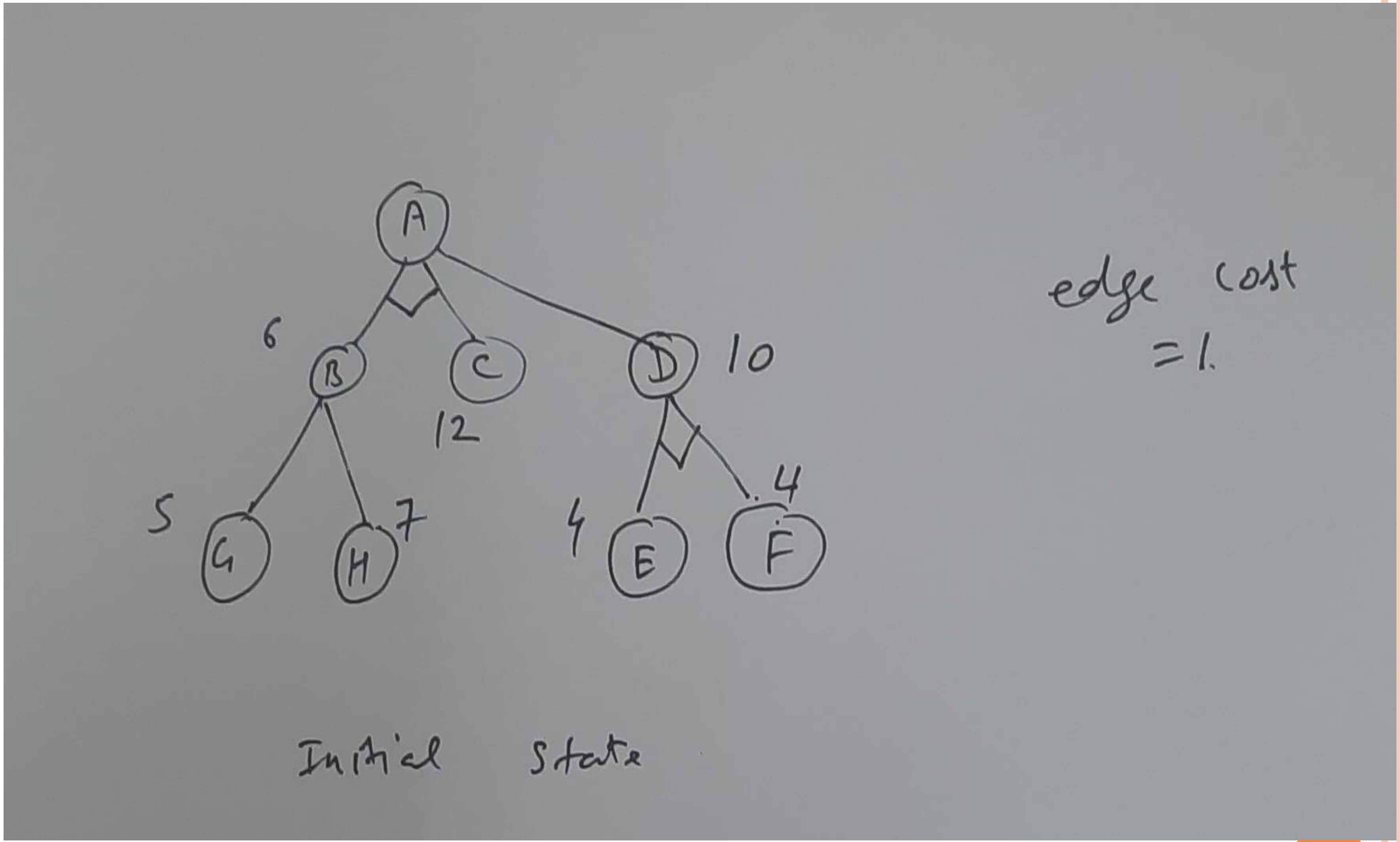


Or another one: Theorem proving in which we reason backwards from the theorem we're trying to prove.

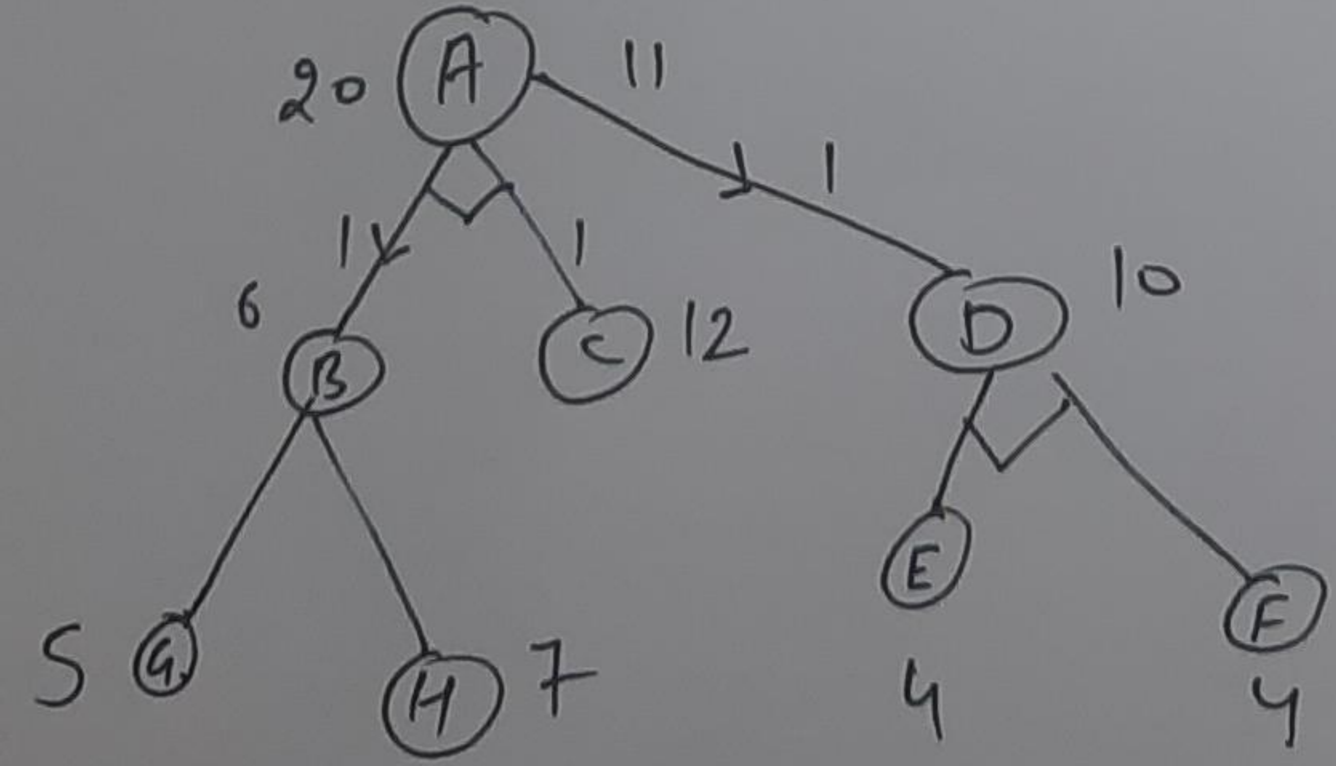
Hyper edge


want to eat food
order from restaurant
purchase ingredients
cook
AND
OR

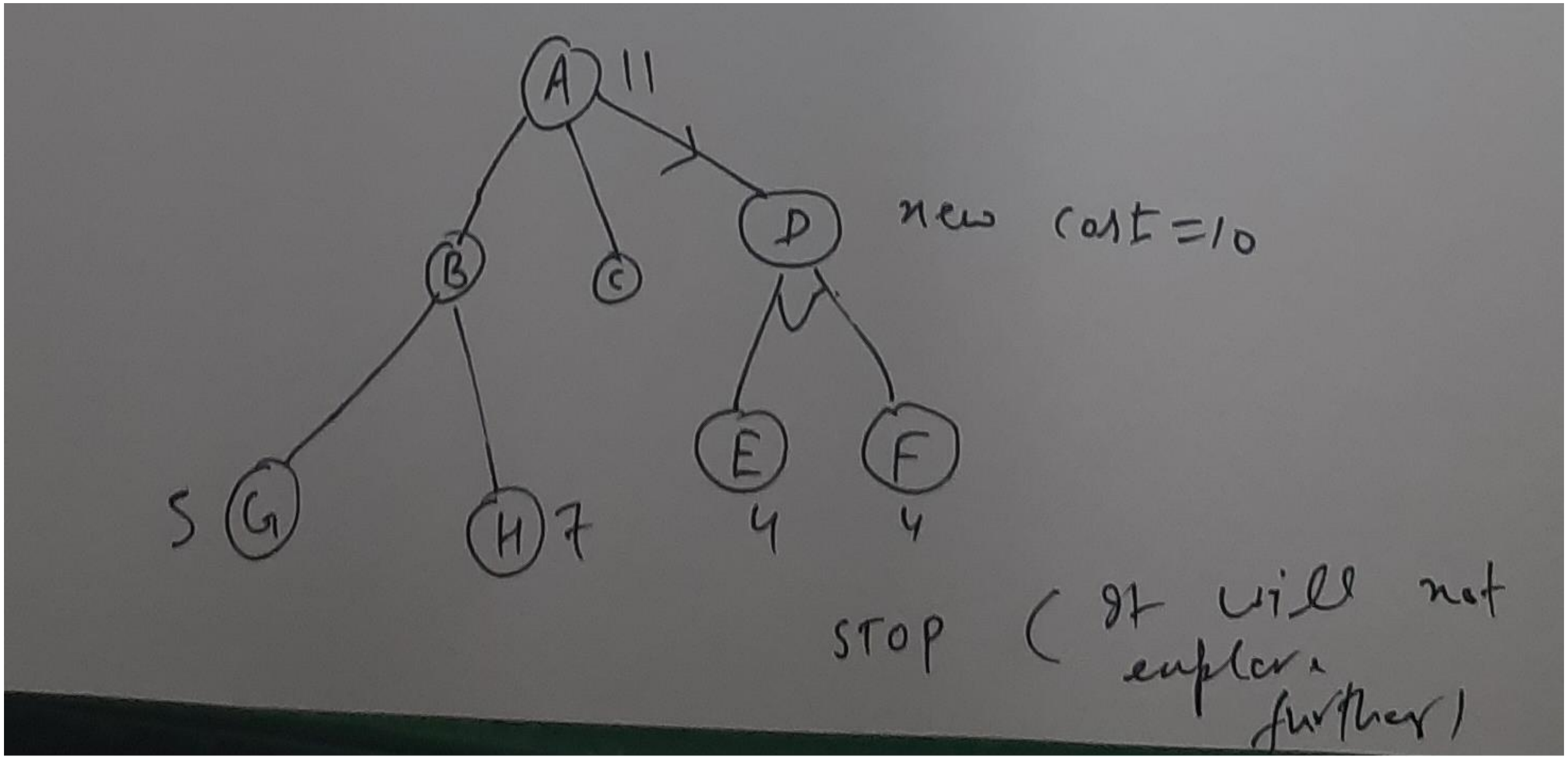
Example 1

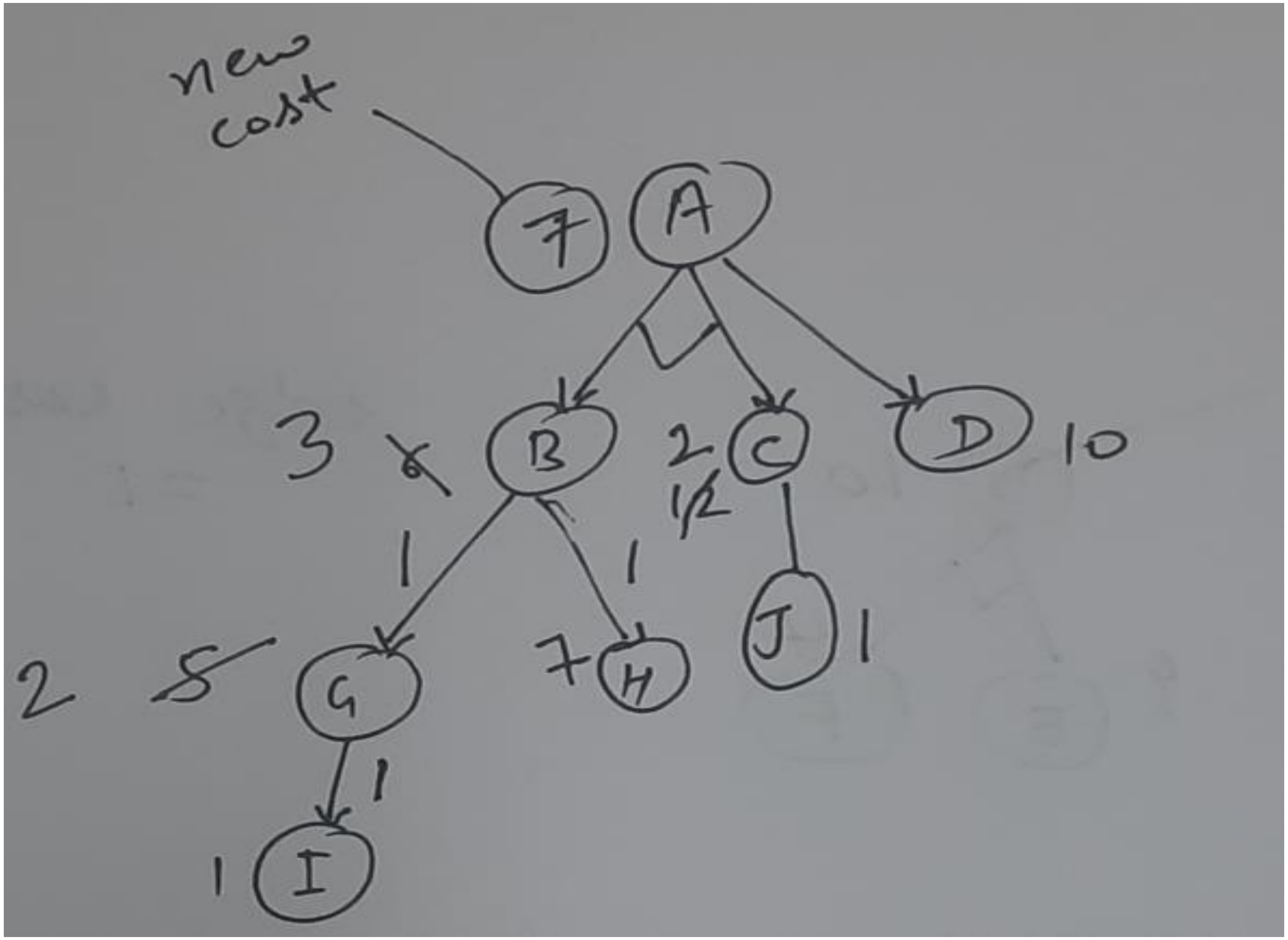


6 is the Heuristic value of B.



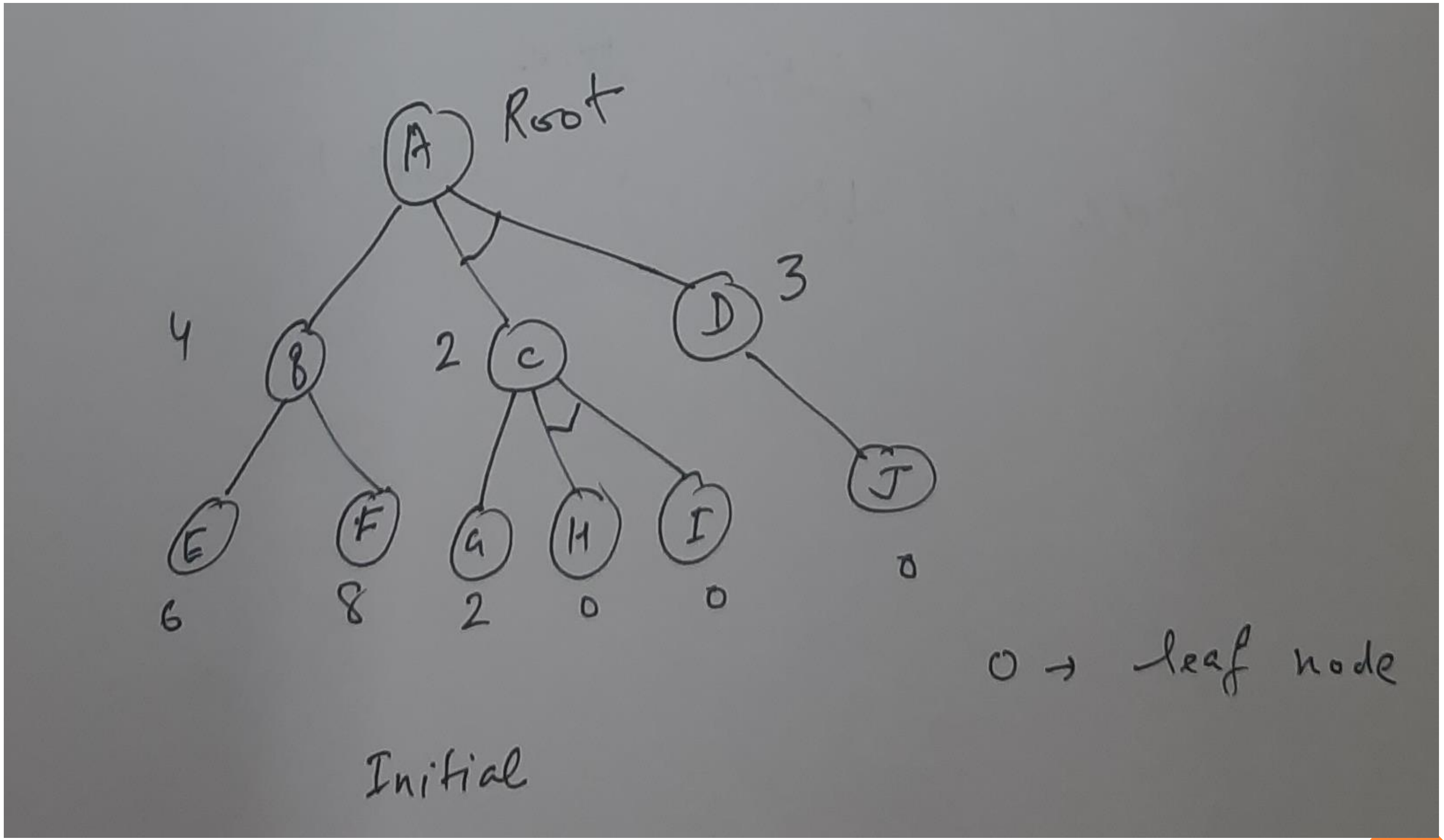
we will select 11


A 11
new cost = 10
B
C
D
5 G
H 7
E
F
4
4
STOP (It will not explore further)

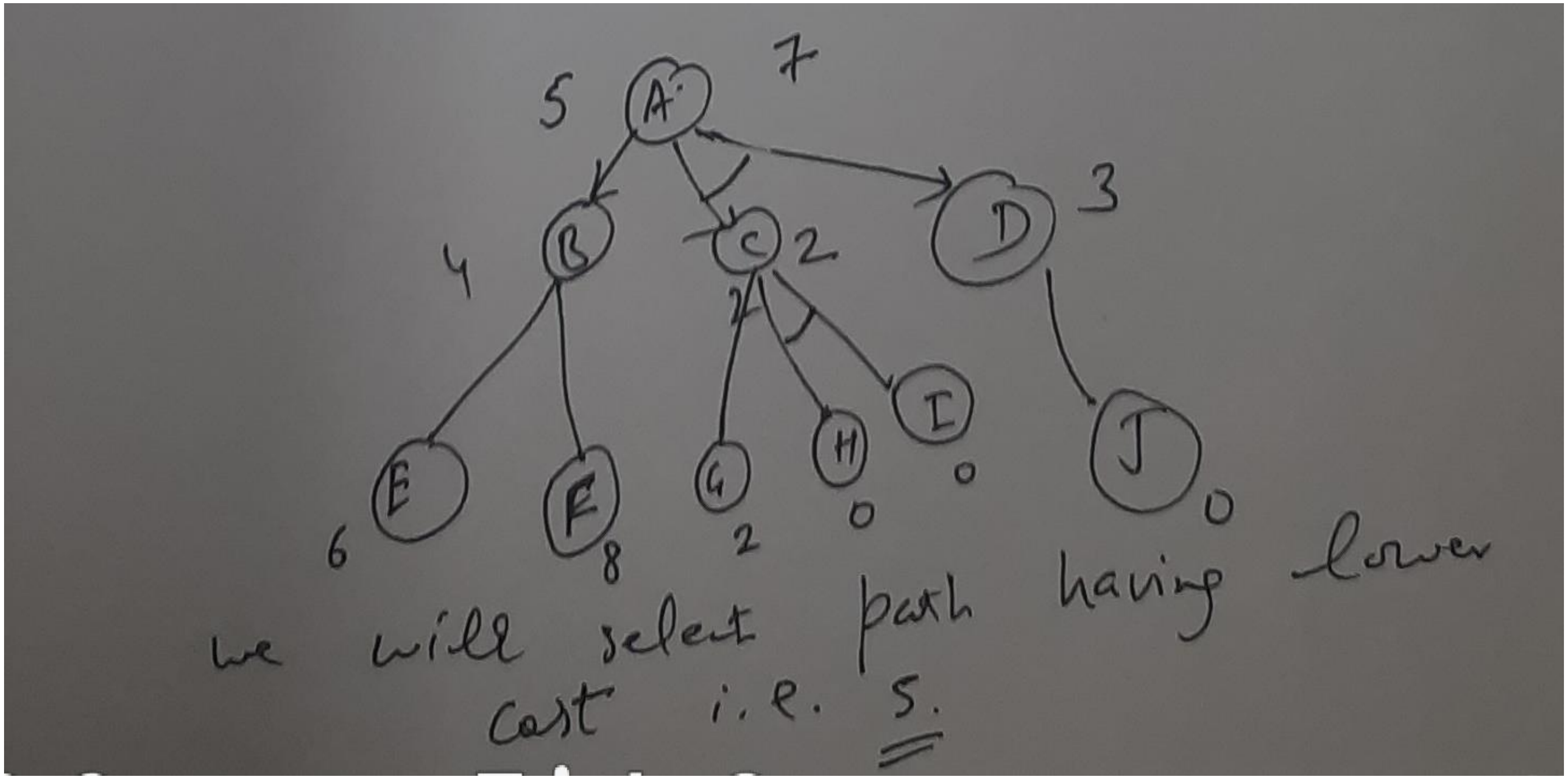


But AO* will not explore this path.

Example II

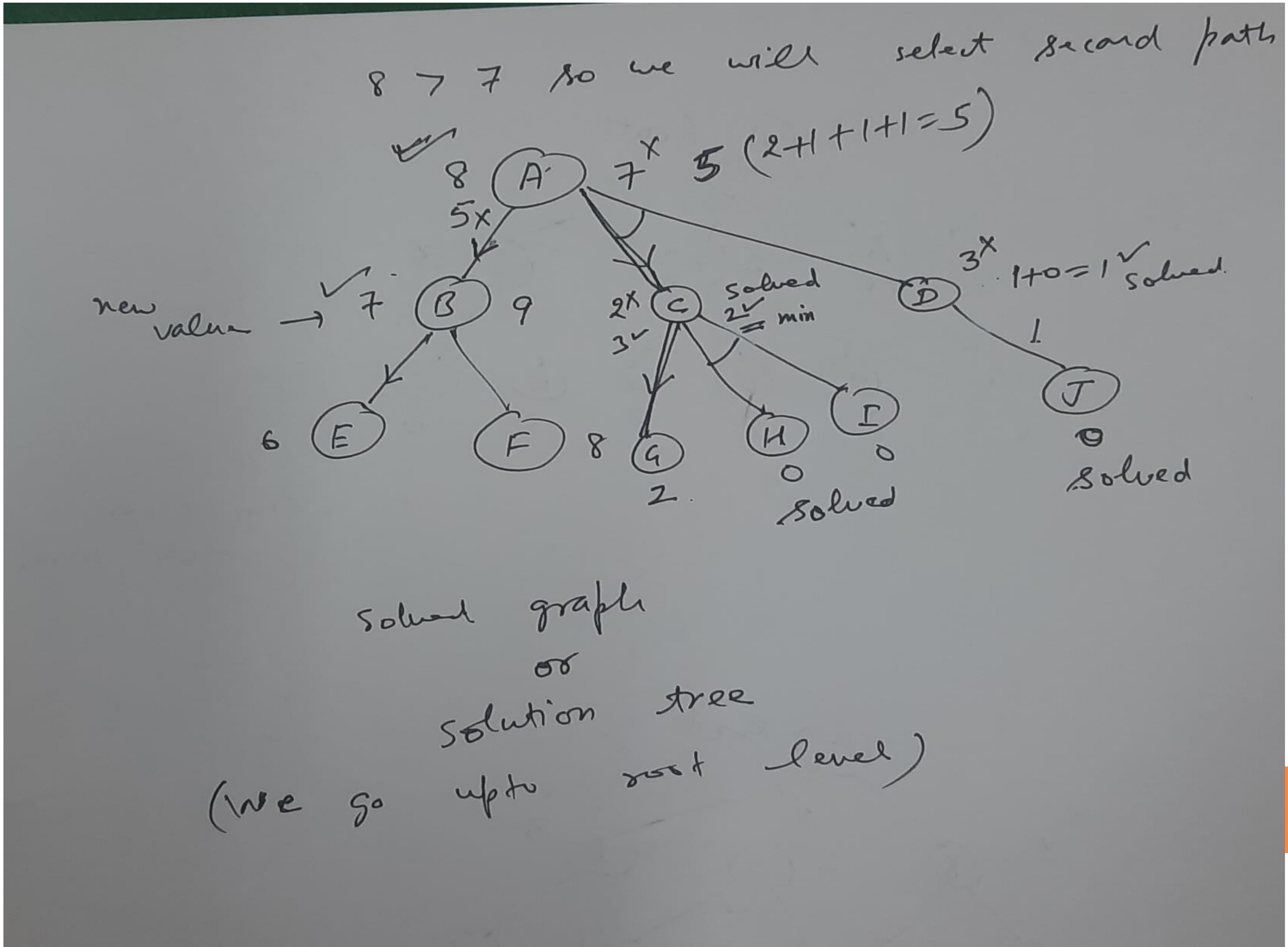


Example II



A
5
7
B
4
C
2
D
3
E
6
F
8
G
2
H
0
I
0
J
0
we will select path having lower cost i.e. 5.

Example II



8 > 7 so we will select second path
8
A
7 5 (2+1+1+1=5)
5
B
7
9
new value →
C
2 3
solved 2 min
D
3
1+0=1 solved
1
E
6
F
8
G
2
H
0
I
0
solved
J
0
solved
solved graph or solution tree (we go upto root level)

Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- There might be some situations where more than one agent is searching for the solution in the same search space.
- The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Mini-Max Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Min-Max algorithm is mostly used for game playing in AI such as Chess, Checkers, tic-tac-toe, go, and various other games. This algorithm computes the minimax decision for the current state.
- In this algorithm, two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while it should get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

Working of Min-Max Algorithm:

- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Utility

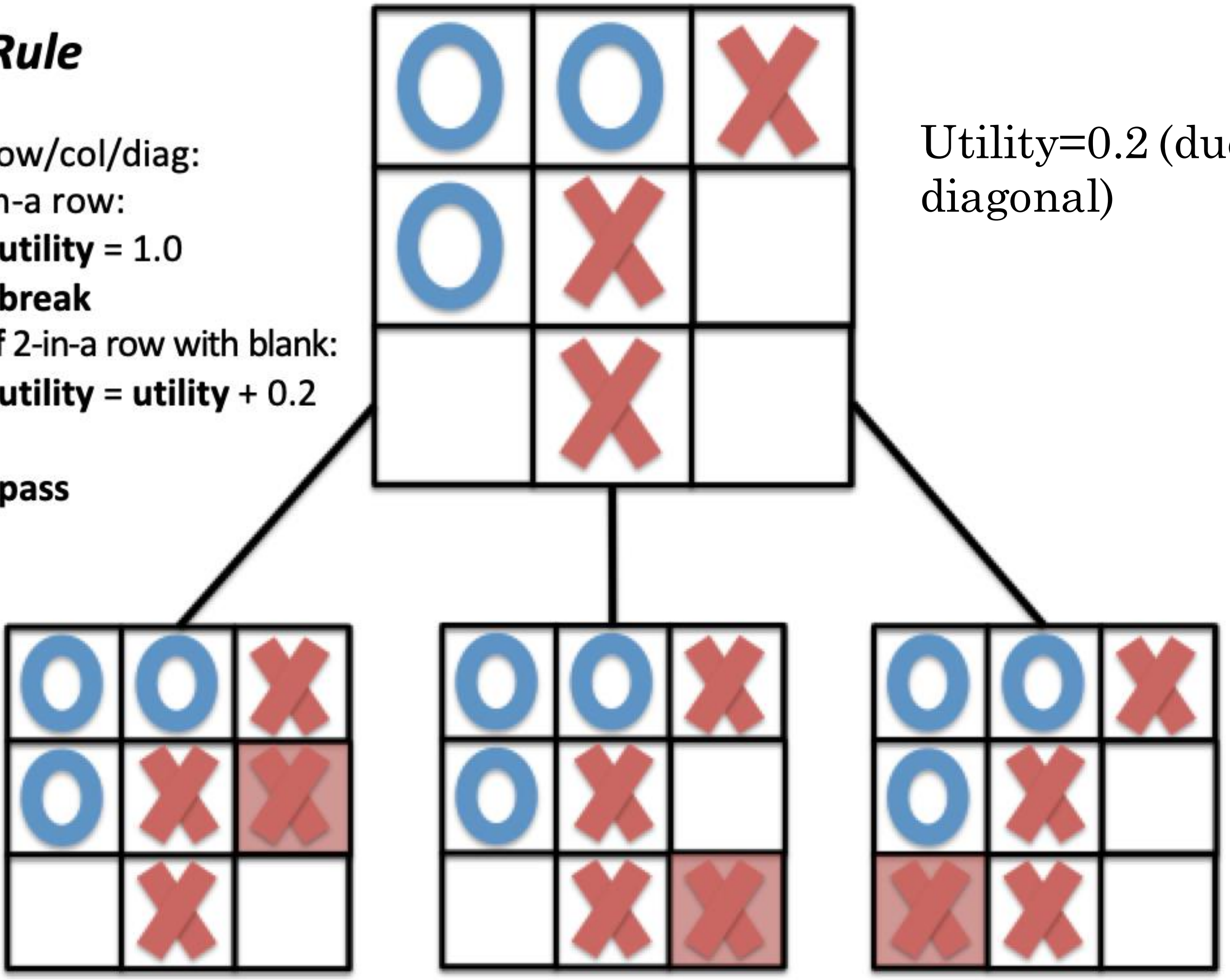
Utility can be thought of as a way to "score" each possible move based on its potential to result in a win.

For example, number of blank spaces on the board, the location of the opponent's current pieces, the location of our current pieces, how close we are to a winning formation, etc. all might be factors to consider in calculating the utility of a particular move.

Current Board – It's X's turn.

Utility Rule

```
utility = 0
for each row/col/diag:
    if 3-in-a row:
        utility = 1.0
        break
    else if 2-in-a row with blank:
        utility = utility + 0.2
    else:
        pass
```

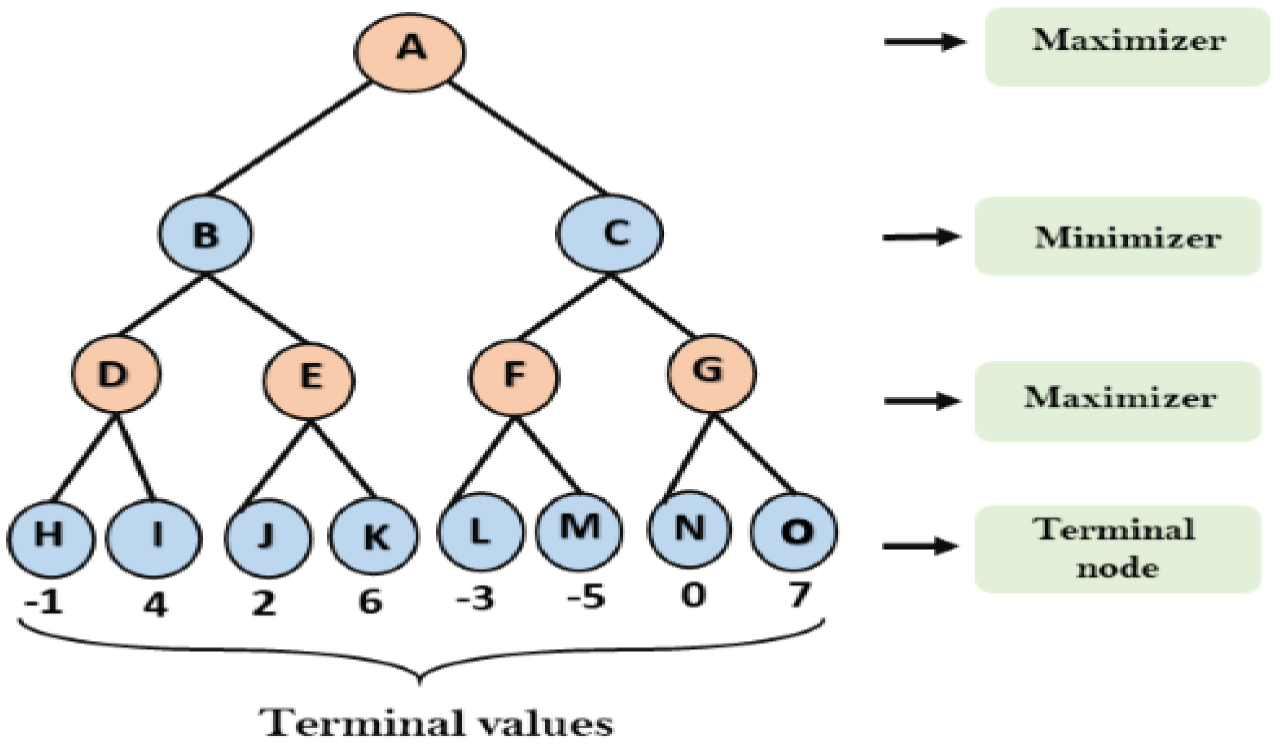


Utility=0.2 (due to diagonal)

utility = 0.4

utility = 0.6

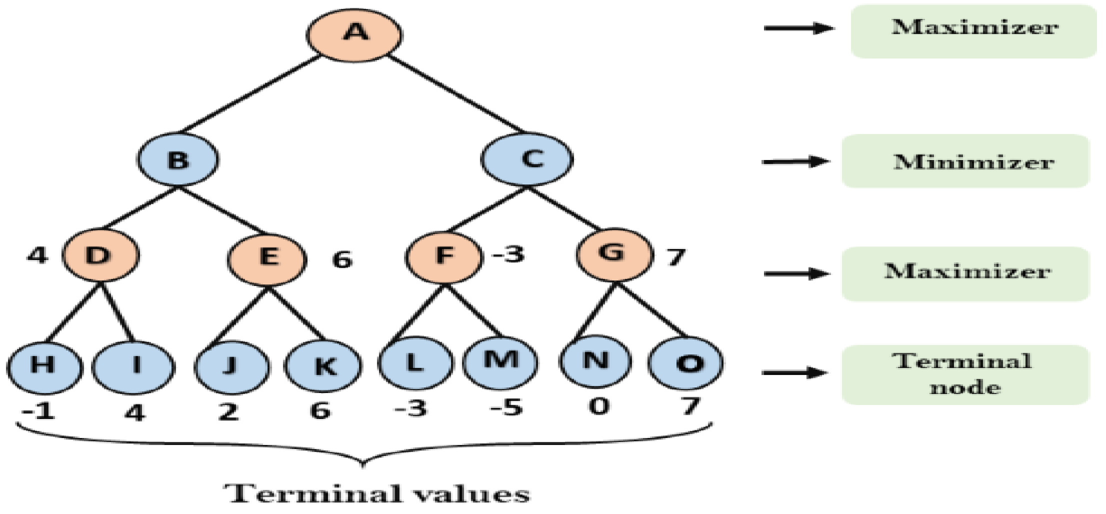
utility = 1.0 !


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
-1
4
2
6
-3
-5
0
7
Terminal values
Maximizer
Minimizer
Maximizer
Terminal node

Step-1: In the first step, the algorithm generates the entire game-tree and applies the utility function to get the utility values for the terminal states. In the tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value = - infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

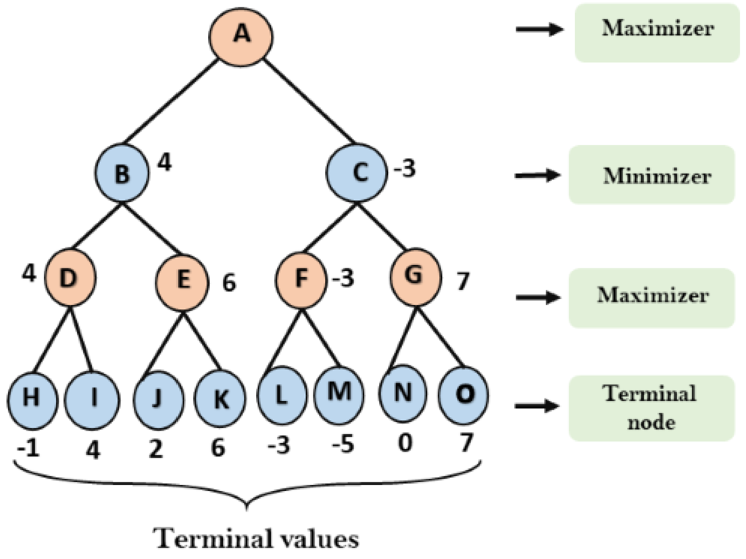
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

For node D	max(-1,- -∞) => max(-1,4)= 4
For Node E	max(2, -∞) => max(2, 6)= 6
For Node F	max(-3, -∞) => max(-3,-5) = -3
For node G	max(0, -∞) = max(0, 7) = 7



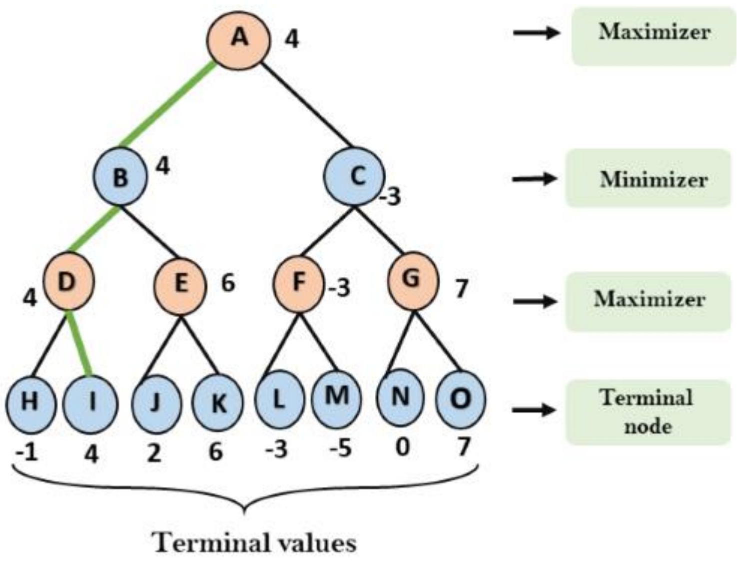
Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

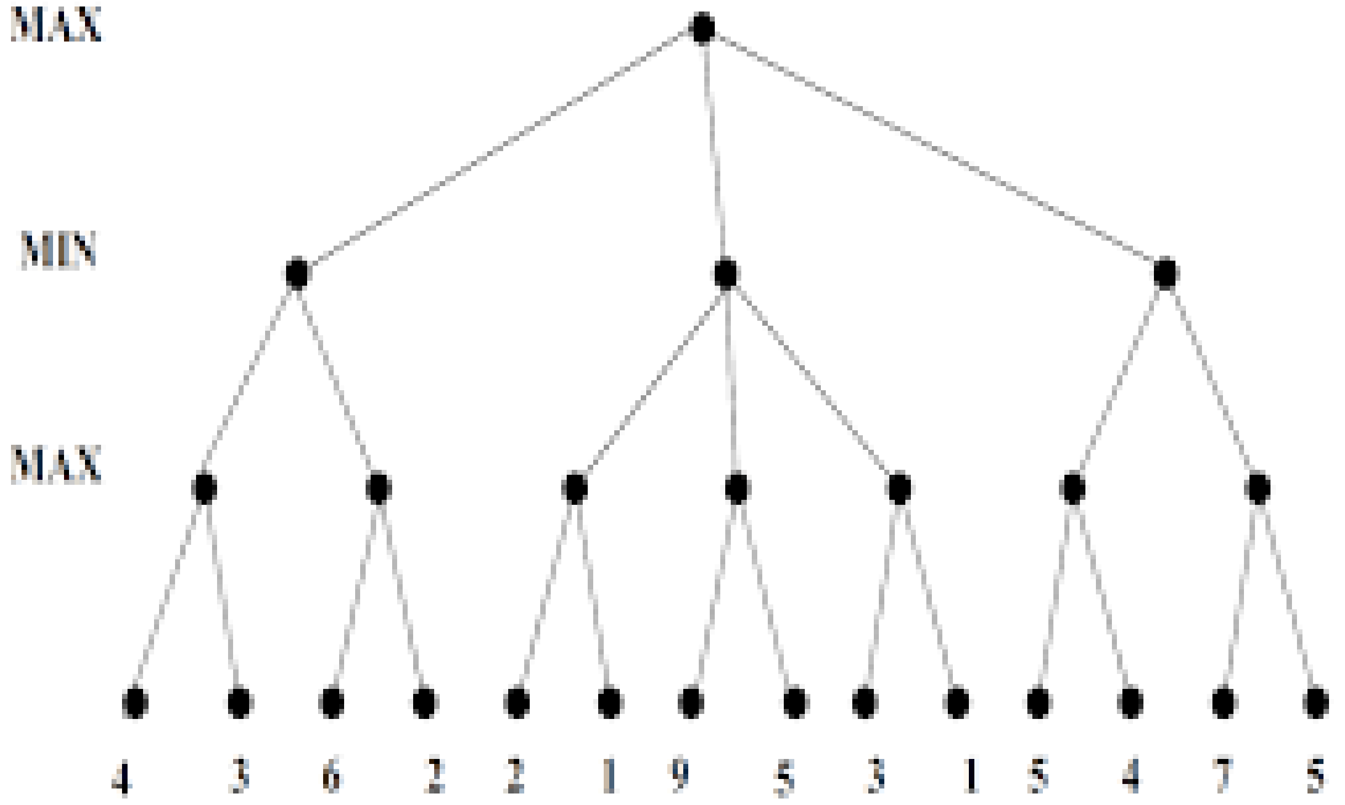
For node B= min(4,6) = 4
For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

For node A max(4, -3)= 4




MAX
MIN
MAX
4
3
6
2
2
1
9
5
3
1
5
4
7
5

Time complexity : $O(b^d)$
Where

- b is the branching factor
- d is the depth of the search tree.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it is very slow for complex games such as Chess, go, etc. These type of games have a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning

Alpha-Beta Pruning

- Alpha-beta pruning is an optimized version of the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of 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.

Alpha-Beta Pruning

- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prunes the tree leaves but also entire sub-tree.
- These parameters 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 +∞.

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.

Condition for Alpha-beta pruning:

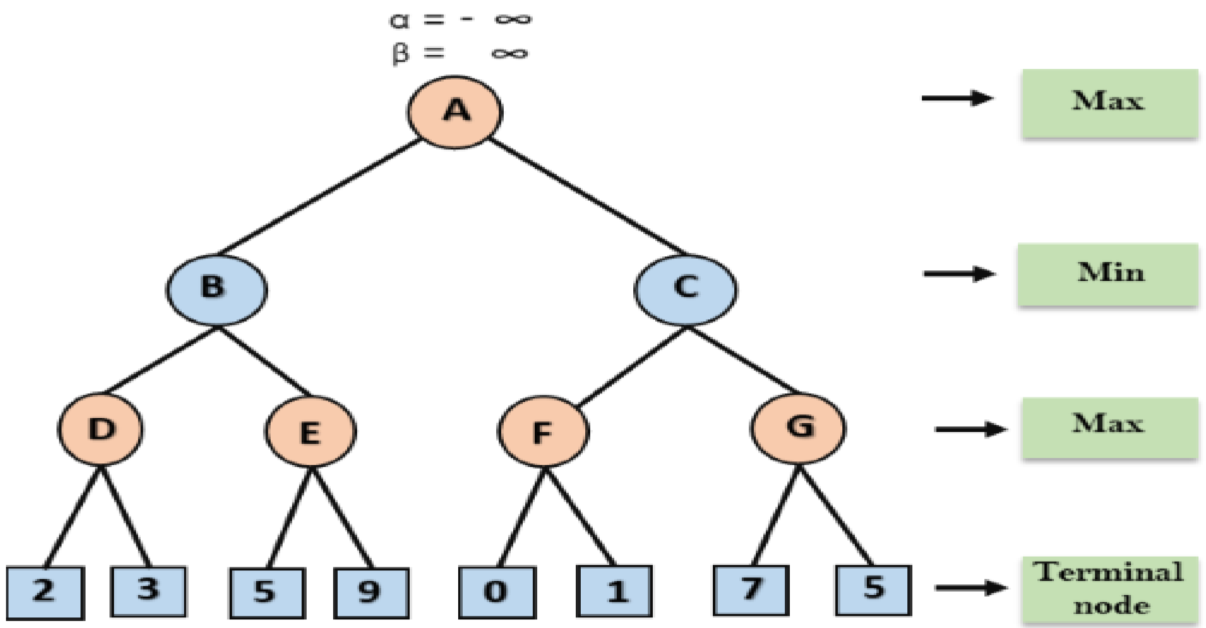
The main condition which required for alpha-beta pruning is:

$\alpha >= \beta$

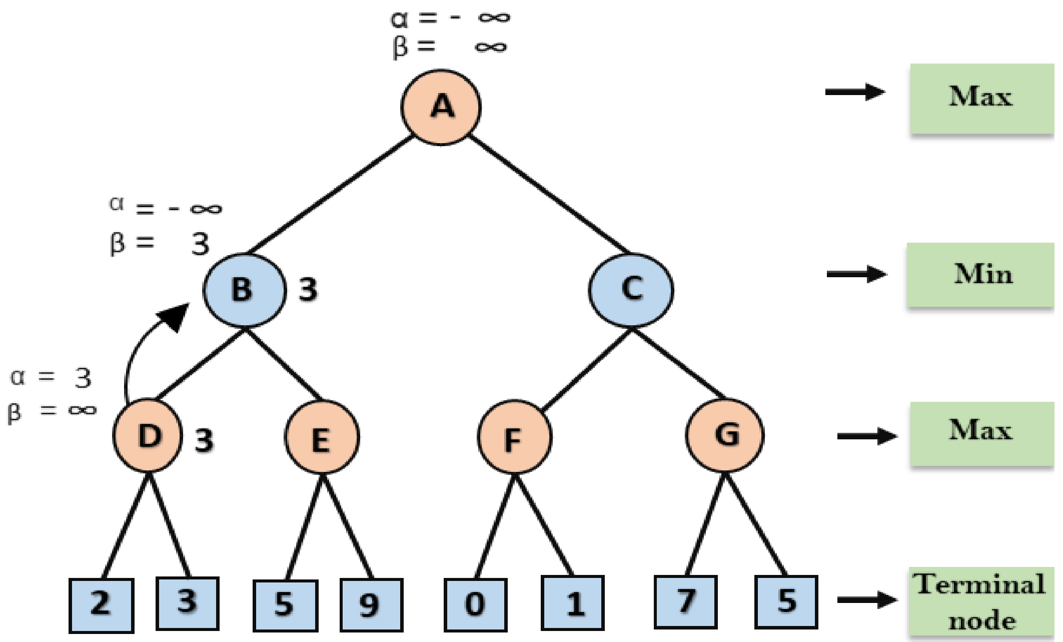
Working of Alpha-Beta Pruning

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 values of alpha and beta are passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

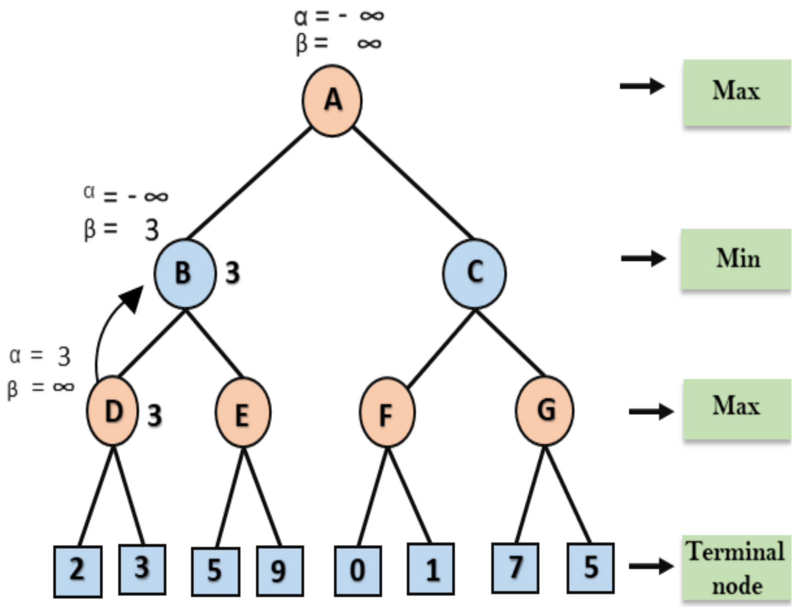


Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

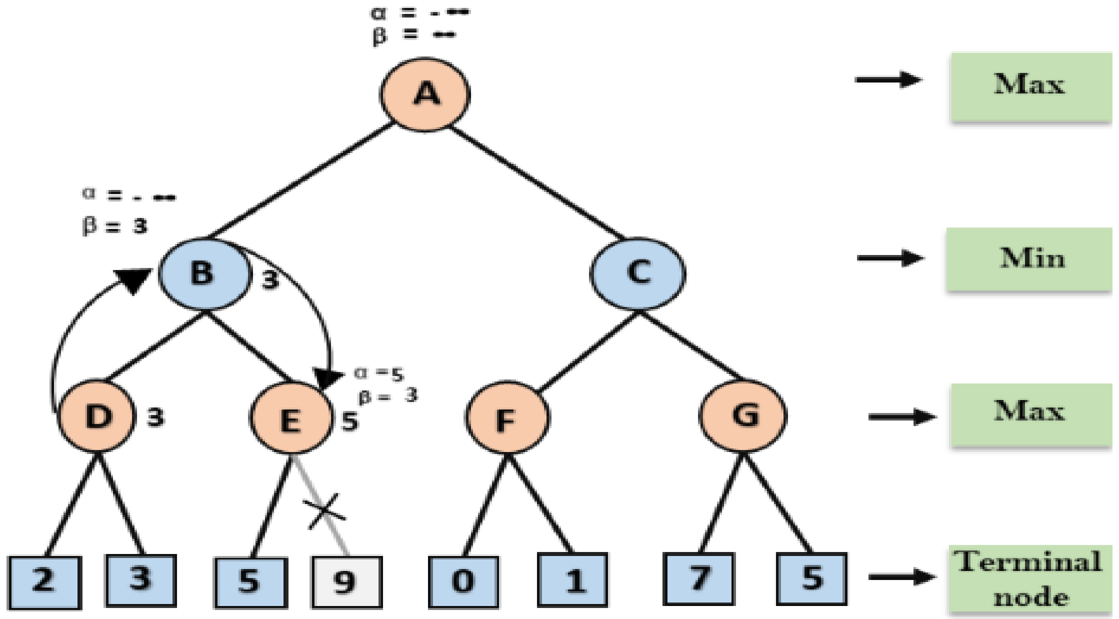


Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, is compared with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

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

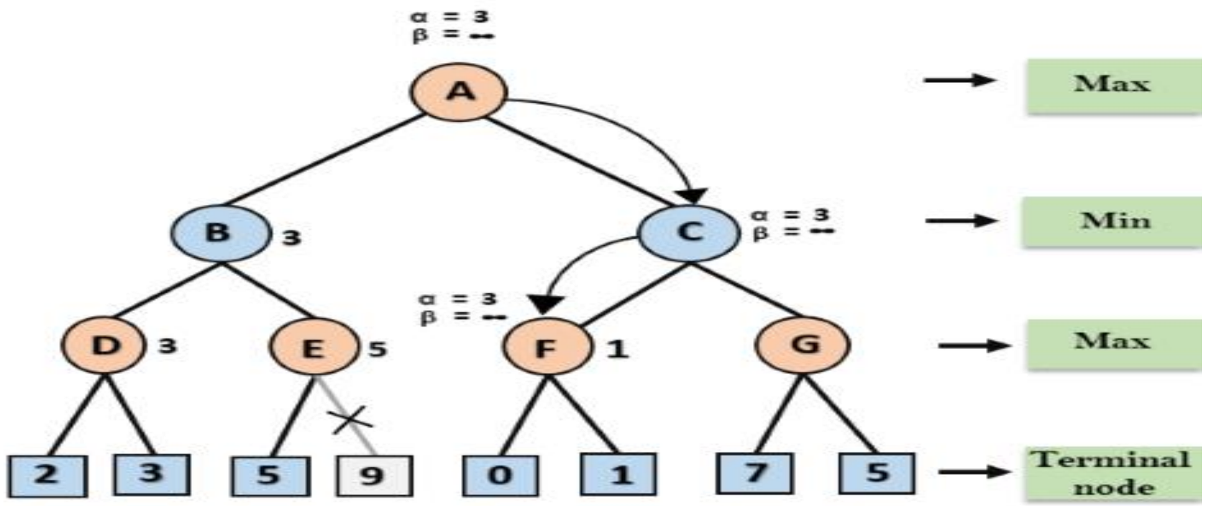


Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5

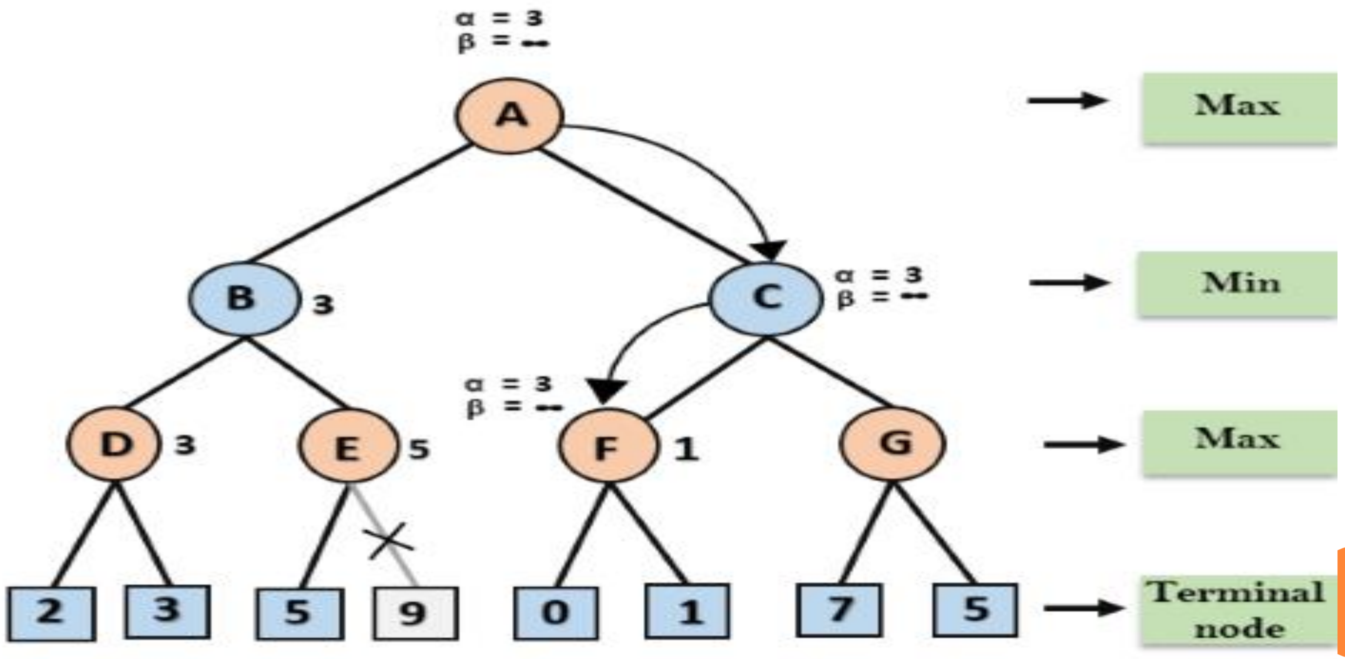


Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed to 3 as max (-∞, 3)= 3, and β= +∞, these two values now passed to right successor of A which is Node C.

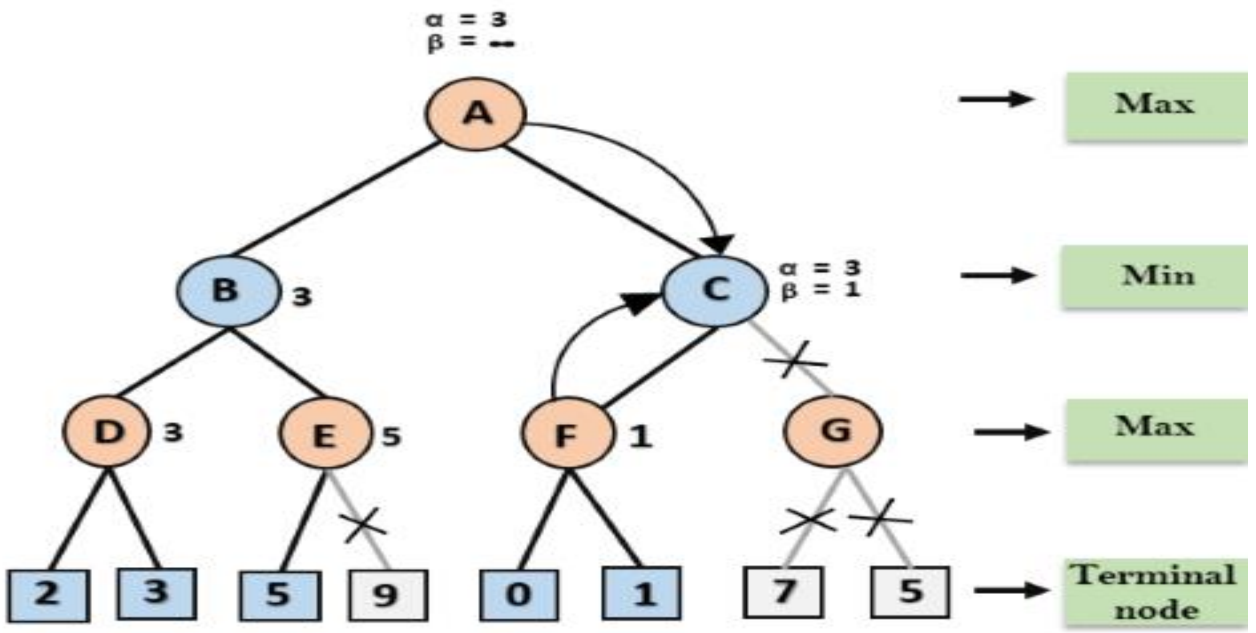
At node C, α=3 and β= +∞, and the same values will be passed on to node F.



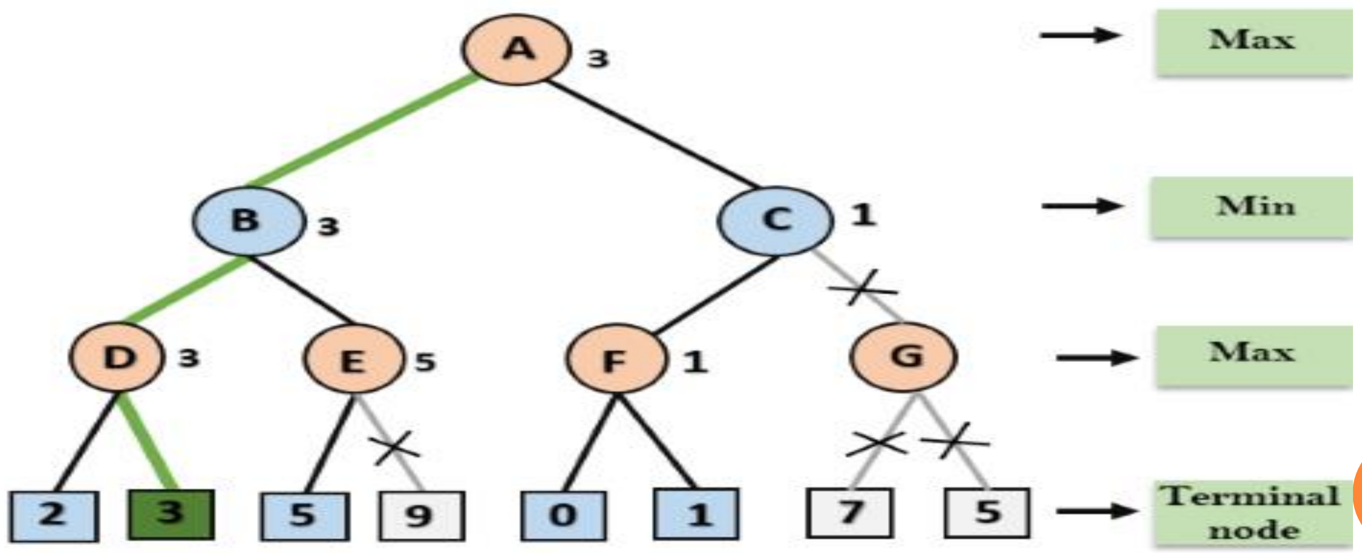
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1



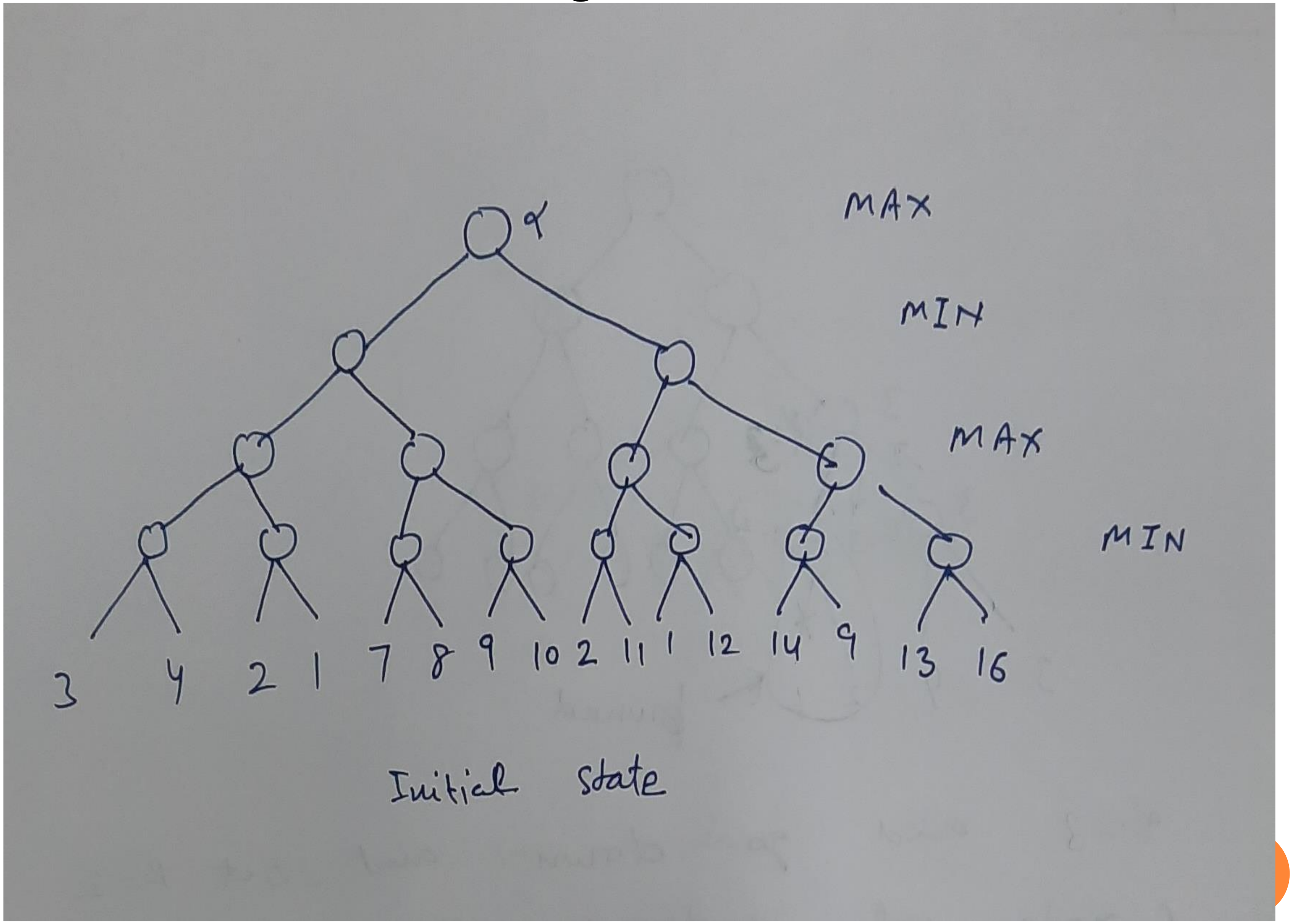
Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



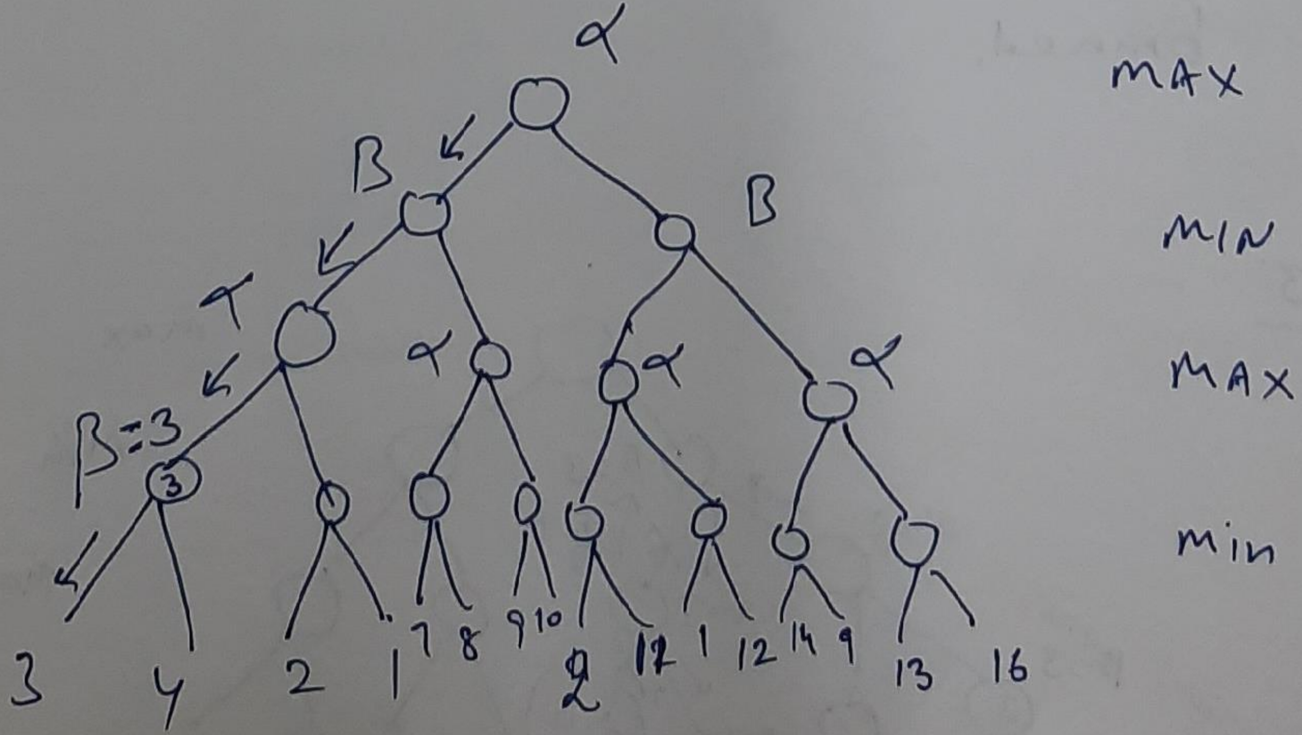
Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Assignment

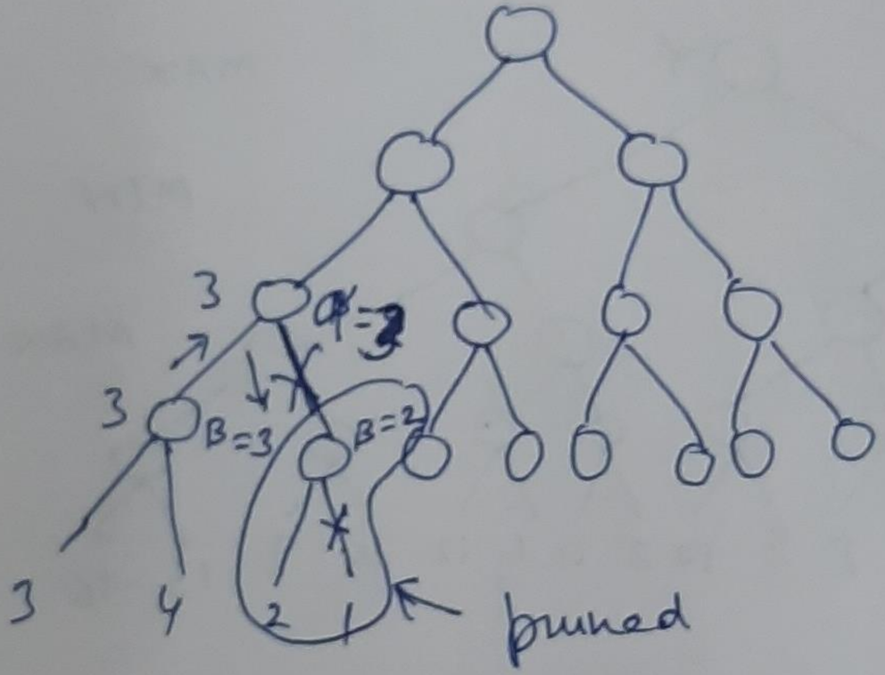


Step I :-



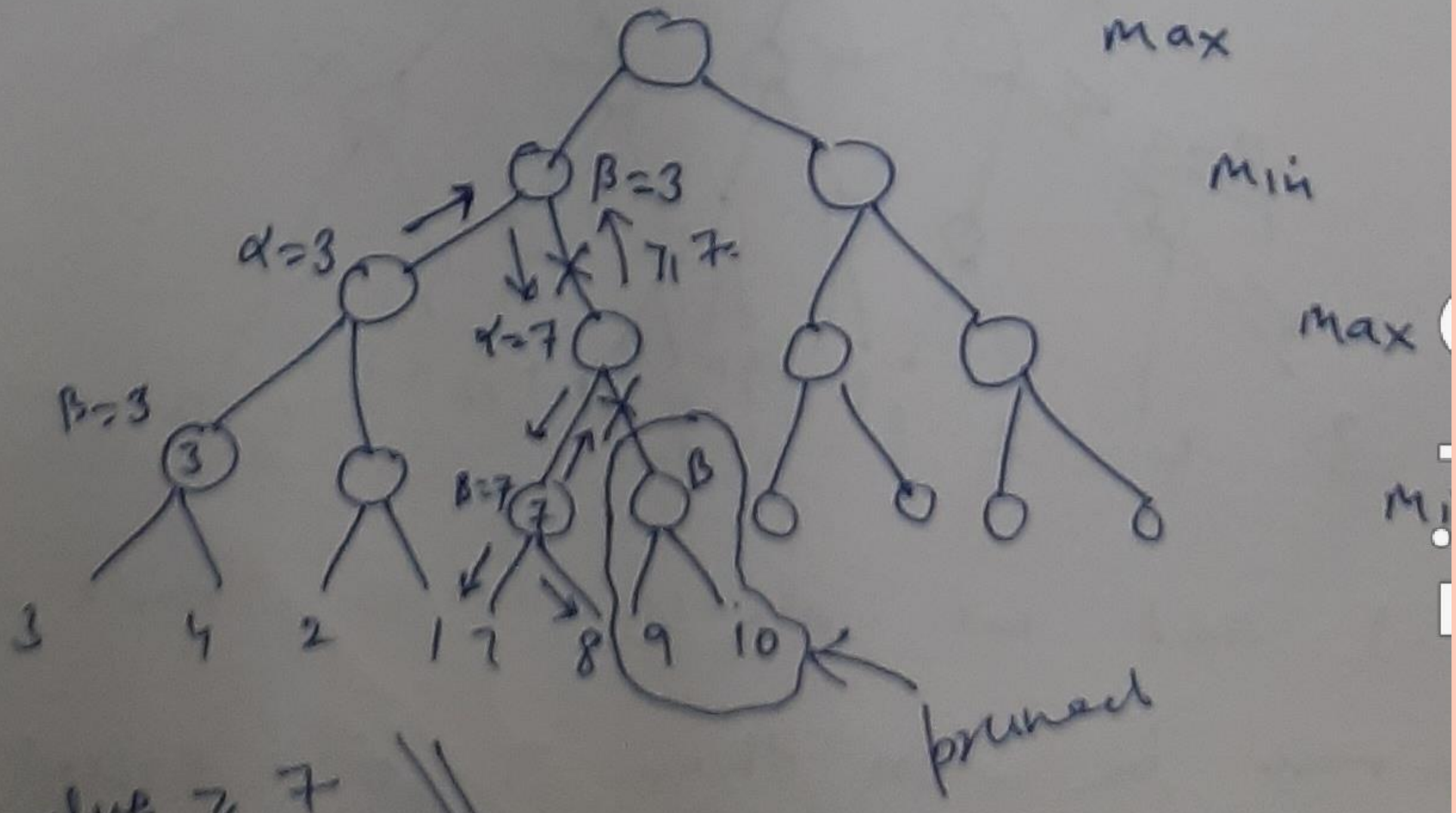
β=3 and we don't know node value (It may be less than 3 or equal to 3) but it is less 4, so β=3

Step 2 :—



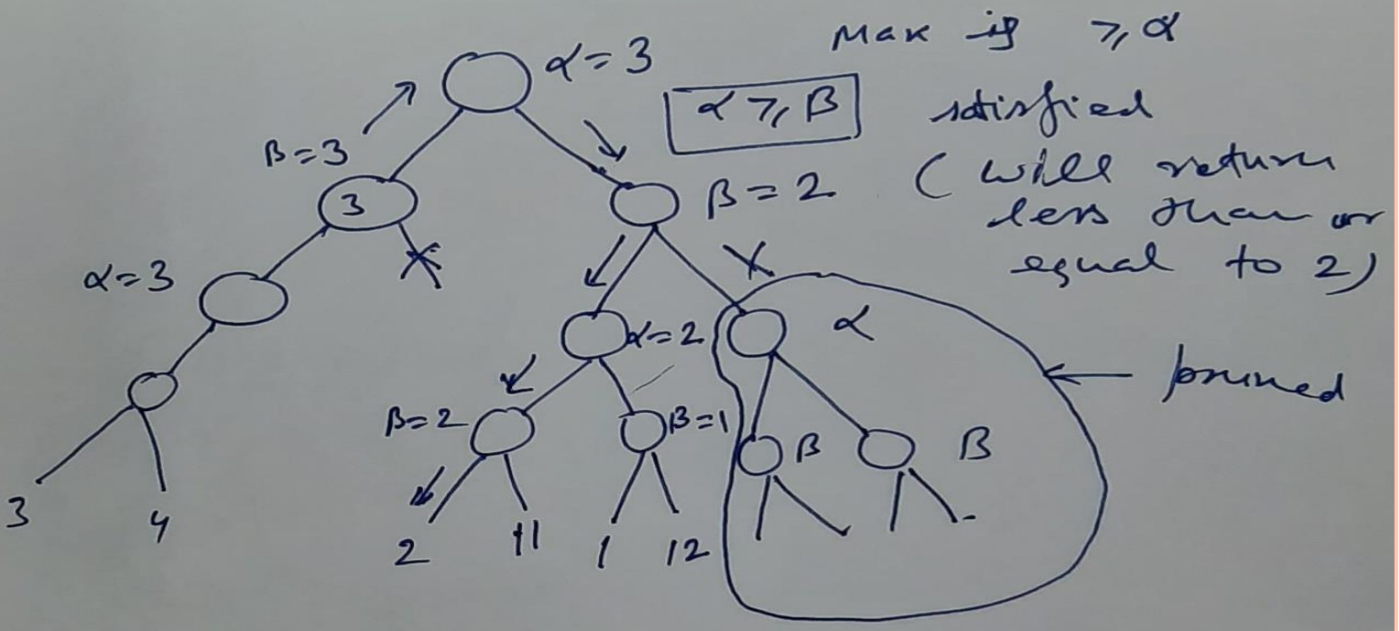
$\alpha=3$ and go down and set $B=2$ (node value will be less than or equal to 2) $\alpha \geq B$, so the subtree is pruned.

Step 3 :-



max value ≥ 7
but β = 3

Step 4 :—



Root node is max which has $\alpha = 3$ and on the right subtree we will get β value ≤ 2. So in every case this right subtree will not benefit the root node (i.e. α value). Right subtree is pruned.

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.

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 $O(b^d)$.

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 $O(b^{d/2})$.