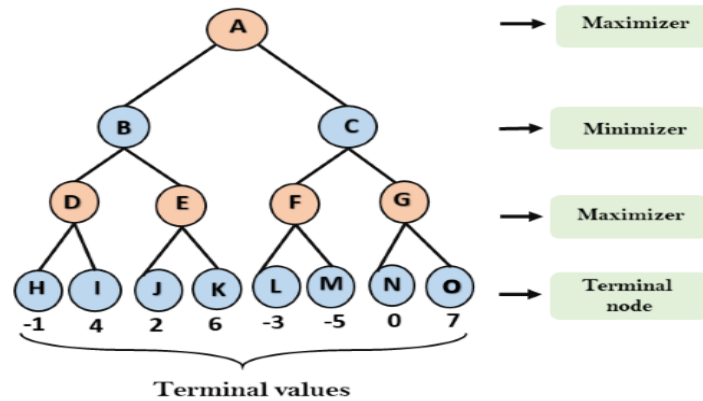


## Lab-6

PRELAB:

1. Write the complete workflow of the minimax two player game.

lab-6190031187  
Radhakrishnaprelab

1. The working of the minimax algorithm can be easily described using an example

Below we have taken an example of game-tree which is representing the two-player game.

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 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 these value and backtrack the tree until initial state occurs

For D  $\max(-1, 4) = 4$

E  $\max(2, 6) = 6$

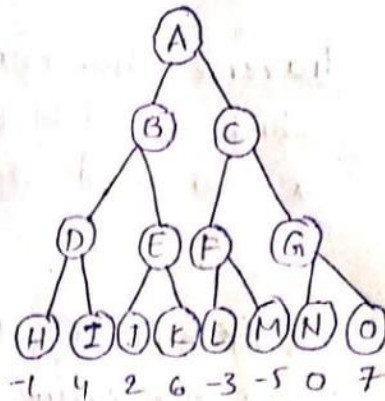
F  $\max(-3, -5) = -3$

G  $\max(0, 7) = 7$

For B  $\min(4, 6) = 4$

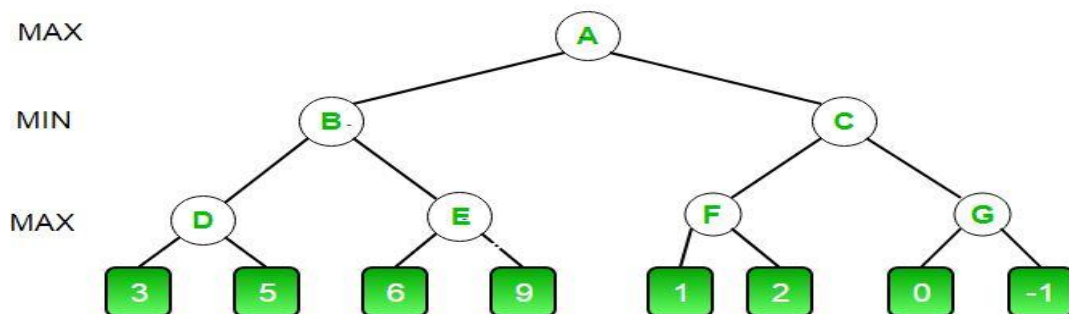
C  $\min(-3, 7) = -3$

For A  $\max(4, -3) = 4$



That was the complete work flow of the minimax two player game.

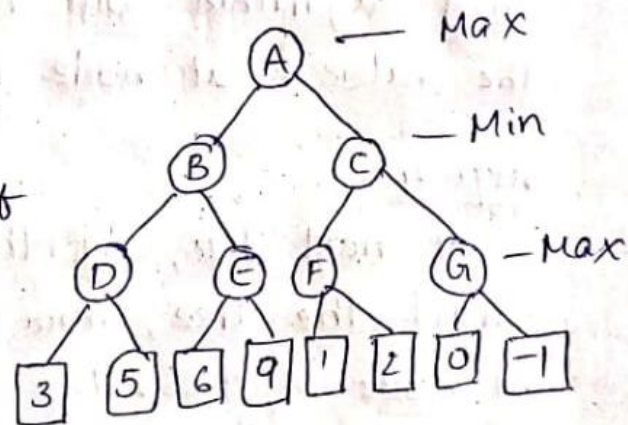
2. Write the complete workflow of the tree using alpha-beta Pruning.



2. Step-1

At D,  $\max(3, 5) = 5$

5 will be the value of  $\alpha$  at node D and node value will also 5

Step-2

Now algorithm backtrack to node B, where the value of  $\beta$  will change as this is a turn of min, Now  $\beta = +\infty$ , will compare with available subsequent node values i.e.  $\min(\infty, 5) = 5$ , hence at node B now  $\alpha = -\infty$  and  $\beta = 5$ . In next step, algorithm

traverse the next successor of Node B which is node E, and the values of  $\alpha = -\infty$  and  $\beta = 5$  also be passed

Step-3

At node E, Max will takes its turn and the value of alpha will change. The current value of alpha will be compared with 6, so  $\max(-\infty, 6) = 6$ , hence at node E  $\alpha = 6$  and  $\beta = 5$ , where  $\alpha \geq \beta$ , so the



right successor of E will be pruned and algorithm will not traverse it, and the value at node E will be 6

#### step-4

At next step, algorithm again back-track the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 5 as  $\max(-\infty, 5) = 5$ , and  $\beta = +\infty$ , these two values now passes to right successor of A which is node C.

At node C,  $\alpha = 5$ ,  $\beta = +\infty$ , and the same values will be passed on to node F.

#### step-5

At node F, again the value of  $\alpha$  will be compared with right child which is 2 and  $\max(5, 2) = 5$  still  $\alpha$  remains 5, but the node value of F will become 2

#### step-6

Node F returns the node value 2 to node C, at C  $\alpha = 5$ ,  $\beta = +\infty$ , here the value of beta will be changed, it will compare with 2 so  $\min(\infty, 2) = 2$ . Now at C,  $\alpha = 5$  and  $\beta = 2$ , and again it satisfies the condition  $\alpha \geq \beta$ , so the next child of C which is G will be pruned, and the algorithm will not compute the entire subtree G.

#### step-7

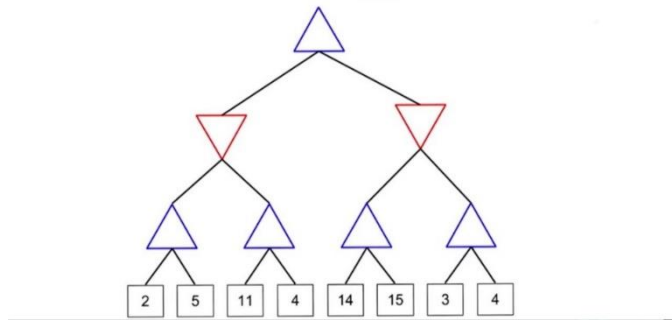
C now returns the value of 2 to A here the best value for A is  $\max(5, 2) = 5$ .

Following is in the final game tree which is showing the nodes which are computed and nodes which has never computed.

Hence the optimal value for the maximizer is 5 for this example.

**INLAB:**

1. Write a python code to print the root node using minimax algorithm.



Output:

The Root Node value is: 5

**CODE**

```
import numpy as np
class Game_Tree:

    succs=dict(A=dict(a1='B',a2='C'),
               B=dict(b1='D',b2='E'),
               C=dict(c1='F',c2='G'),
               D=dict(d1='D1',d2='D2'),
               E=dict(e1='E1',e2='E2'),
               F=dict(f1='F1',f2='F2'),
               G=dict(g1='G1',g2='G2'))
    utils=dict(D1=2,D2=5,E1=11,E2=4,F1=14,F2=15,G1=3,G2=4)

    def actions(self,state):
        return list(self.succs.get(state,{}).keys())

    def result(self,state,move):
        return self.succs[state][move]

    def utility(self,state,player):
        if player=='MAX':
            return self.utils[state]
        else:
            return -self.utils[state]

    def terminal_test(self,state):
        return state not in ('A','B','C','D','E','F','G')

    def to_move(self,state):
        return 'MIN' if state in 'BC' else 'MAX'

    def minmax_decision(state,game):
        player=game.to_move(state)
        def max_value(state):
            if game.terminal_test(state):
                return game.utility(state,player)
            v=-np.inf
```

```

for a in game.actions(state):
    v=max(v,min_value(game.result(state,a)))
return v

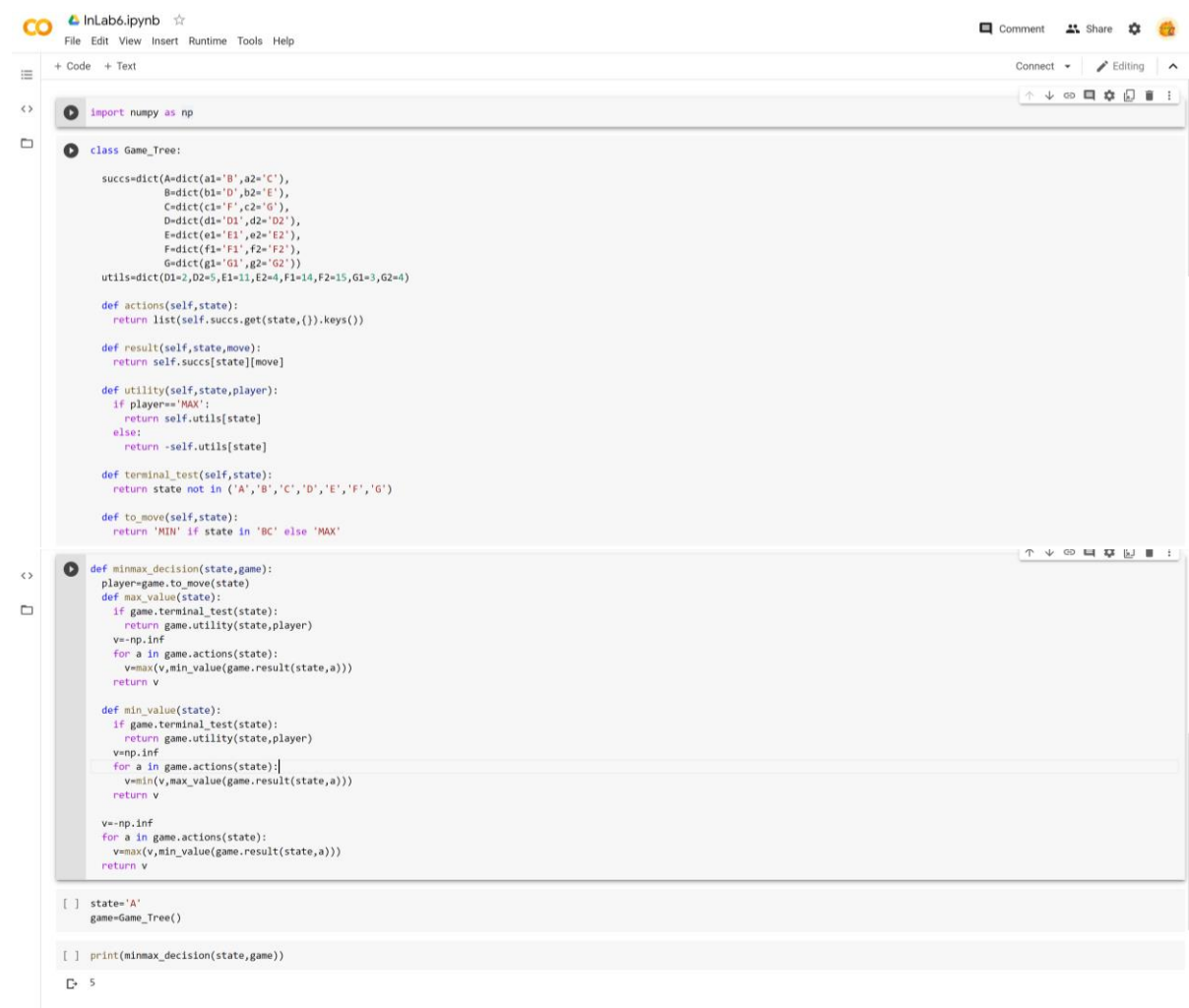
def min_value(state):
    if game.terminal_test(state):
        return game.utility(state,player)
    v=np.inf
    for a in game.actions(state):
        v=min(v,max_value(game.result(state,a)))
    return v

v=-np.inf
for a in game.actions(state):
    v=max(v,min_value(game.result(state,a)))
return v

state='A'
game=Game_Tree()

print(minmax_decision(state,game))

```



The screenshot shows a Jupyter Notebook with the following code:

```

import numpy as np

class Game_Tree:
    succs=dict(A=dict(a1='B',a2='C'),
               B=dict(b1='D',b2='E'),
               C=dict(c1='F',c2='G'),
               D=dict(d1='D1',d2='D2'),
               E=dict(e1='E1',e2='E2'),
               F=dict(f1='F1',f2='F2'),
               G=dict(g1='G1',g2='G2'))
    utils=dict(D1=2,D2=5,E1=1,E2=4,F1=14,F2=15,G1=3,G2=4)

    def actions(self,state):
        return list(self.succs.get(state,{}).keys())

    def result(self,state,move):
        return self.succs[state][move]

    def utility(self,state,player):
        if player=='MAX':
            return self.utils[state]
        else:
            return -self.utils[state]

    def terminal_test(self,state):
        return state not in ('A','B','C','D','E','F','G')

    def to_move(self,state):
        return 'MIN' if state in 'BC' else 'MAX'

def minmax_decision(state,game):
    player=game.to_move(state)
    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state,player)
        v=-np.inf
        for a in game.actions(state):
            v=max(v,min_value(game.result(state,a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state,player)
        v=np.inf
        for a in game.actions(state):
            v=min(v,max_value(game.result(state,a)))
        return v

    v=-np.inf
    for a in game.actions(state):
        v=max(v,min_value(game.result(state,a)))
    return v

[ ] state='A'
game=Game_Tree()

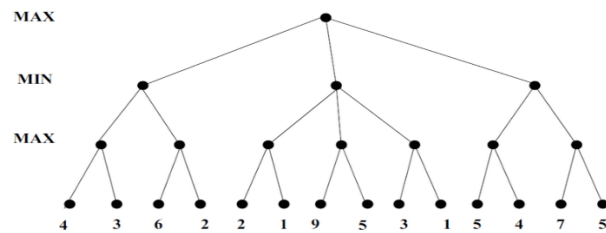
[ ] print(minmax_decision(state,game))

```

At the bottom of the notebook, there is a status bar showing a cursor icon and the number 5.

**POSTLAB:**

1. Write a python code to implement the given tree and print the alpha - beta values in the root node and also print the result and time pruned by using Alpha beta pruning.



Output:

(Alpha, beta): 5 15

Result: 5

Times pruned: 4

**CODE**

```
import numpy as np

class Game_Tree:
    succs=dict(A=dict(a1='B',a2='C',a3='D'),
               B=dict(b1='E',b2='F'),
               C=dict(c1='G',c2='H',c3='I'),
               D=dict(d1='J',d2='K'),
               E=dict(e1='E1',e2='E2'),
               F=dict(f1='F1',f2='F2'),
               G=dict(g1='G1',g2='G2'),
               H=dict(h1='H1',h2='H2'),
               I=dict(i1='I1',i2='I2'),
               J=dict(j1='J1',j2='J2'),
               K=dict(k1='K1',k2='K2'))
    utils=dict(E1=4,E2=3,F1=6,F2=4,G1=2,G2=1,H1=9,H2=5,I1=3,I2=1,J1=5,J2=
4,K1=7,K2=5)
    initial='A'

    def actions(self,state):
        return list(self.succs.get(state,{}).keys())
    def result(self,state,move):
        return self.succs[state][move]
    def utility(self,state,player):
        if player=='MAX':
            return self.utils[state]
        else:
            return -self.utils[state]
    def terminal_test(self,state):
        return state not in('A','B','C','D','E','F','G','H','I','J','K')
    def to_move(self,state):
```

```
        return 'MIN' if state in 'BCD' else 'MAX'

p=0
alpha=-np.inf
beta=np.inf
def alpha_beta_search(state,game):
    player=game.to_move(state)

    #Functions used by alpha_beta
    def max_value(state,alpha,beta):
        global p
        if game.terminal_test(state):
            return game.utility(state,player)
        v=-np.inf
        for a in game.actions(state):
            v=max(v,min_value(game.result(state,a),alpha,beta))
            if v>=beta:
                p=p+1
                return v
            alpha=max(alpha,v)
        return v

    def min_value(state,alpha,beta):
        global p
        if game.terminal_test(state):
            return game.utility(state,player)
        v=np.inf
        for a in game.actions(state):
            v=min(v,max_value(game.result(state,a),alpha,beta))
            if v<=alpha:
                p=p+1
                return v
            beta=min(beta,v)
        return (v)

    #Body of alpha_beta_search
    global alpha
    global beta
    best_action=None
    for a in game.actions(state):
        v=min_value(game.result(state,a),alpha,beta)
        if v > alpha:
            alpha=v
            best_action=a
        else:
            beta=v
    return (v)
```



```

state='A'
game=Game_Tree()

print("Result- ",alpha_beta_search(state,game))
print("Times pruned- ",p)
print("Alpha- ",alpha)
print("Beta- ",beta)

```

Postlab6.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```

import numpy as np

class Game_Tree:
    succs=dict(A=dict(a1='B',a2='C',a3='D'),
              B=dict(b1='E',b2='F'),
              C=dict(c1='G',c2='H',c3='I'),
              D=dict(d1='J',d2='K'),
              E=dict(e1='I1',e2='I2'),
              F=dict(f1='J1',f2='J2'),
              G=dict(g1='K1',g2='K2'),
              H=dict(h1='H1',h2='H2'),
              I=dict(i1='I1',i2='I2'),
              J=dict(j1='J1',j2='J2'),
              K=dict(k1='K1',k2='K2'))
    utils=dict(E1=4,E2=3,F1=6,F2=4,G1=2,G2=1,H1=9,H2=5,I1=3,I2=1,J1=5,J2=4,K1=7,K2=5)
    initial='A'

    def actions(self,state):
        return list(self.succs.get(state,{}).keys())
    def result(self,state,move):
        return self.succs[state][move]
    def utility(self,state,player):
        if player=='MAX':
            return self.utils[state]
        else:
            return -self.utils[state]
    def terminal_test(self,state):
        return state not in('A','B','C','D','E','F','G','H','I','J','K')
    def to_move(self,state):
        return 'MIN' if state in 'BCD' else 'MAX'

p=0
alpha=-np.inf
beta=np.inf
def alpha_beta_search(state,game):
    player=game.to_move(state)

    #Functions used by alpha_beta
    def max_value(state,alpha,beta):
        global p
        if game.terminal_test(state):
            return game.utility(state,player)
        v=-np.inf
        for a in game.actions(state):
            v=max(v,min_value(game.result(state,a),alpha,beta))
            if v>=beta:
                p+=1
                return v
            alpha=max(alpha,v)
        return v

    def min_value(state,alpha,beta):
        global p
        if game.terminal_test(state):
            return game.utility(state,player)
        v=np.inf
        for a in game.actions(state):
            v=min(v,max_value(game.result(state,a),alpha,beta))
            if v<=alpha:
                p+=1
                return v
            beta=min(beta,v)

    return (v)

[3] return (v)

#Body of alpha_beta_search
global alpha
global beta
best_action=None
for a in game.actions(state):
    v=min_value(game.result(state,a),alpha,beta)
    if v > alpha:
        alpha=v
        best_action=a
    else:
        beta=v
return (v)

[4] state='A'
game=Game_Tree()

print("Result- ",alpha_beta_search(state,game))
print("Times pruned- ",p)
print("Alpha- ",alpha)
print("Beta- ",beta)

```

Result- 5  
Times pruned- 4  
Alpha- 5  
Beta- 2