**1] DFS**

graph1={

'A':set(['B','C']),

'B':set(['A','D','E']),

'C':set(['A','F']),

'D':set(['B']),

'E':set(['B','F']),

'F':set(['C','E']),

}

def dfs(graph1,node,visited):

if node not in visited:

visited.append(node)

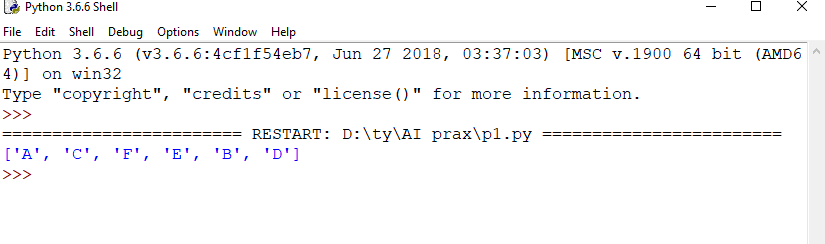
for n in graph1[node]:

dfs(graph1,n,visited)

return visited

visited=dfs(graph1,'A',[])

print(visited)



**2] BFS**

graph = {'A': ['B', 'C'],

'B': ['A','D'],

'C': ['A', 'E','F'],

'D': ['B','B'],

'E': [ 'C','D'],

'F': ['C'],

}

def bfs(graph, start):

explored = []

queue = [start]

levels = {}

levels[start]= 0

visited= [start]

while queue:

node = queue.pop(0)

explored.append(node)

neighbours = graph[node]

for neighbour in neighbours:

if neighbour not in visited:

queue.append(neighbour)

visited.append(neighbour)

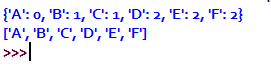
levels[neighbour]= levels[node]+1

print (levels)

return visited

ans = bfs(graph,'A')

print(ans)



3] N-queen Problem

class QueenChessBoard:

def \_\_init\_\_(self, size):

self.size = size

self.columns = []

def place\_in\_next\_row(self, column):

self.columns.append(column)

def remove\_in\_current\_row(self):

return self.columns.pop()

def is\_this\_column\_safe\_in\_next\_row(self, column):

row = len(self.columns)

for queen\_column in self.columns:

if column == queen\_column:

return False

for queen\_row, queen\_column in enumerate(self.columns):

if queen\_column - queen\_row == column - row:

return False

for queen\_row, queen\_column in enumerate(self.columns):

if ((self.size - queen\_column) - queen\_row == (self.size - column) - row):

return False

return True

def display(self):

for row in range(self.size):

for column in range(self.size):

if column == self.columns[row]:

print('Q', end=' ')

else:

print('#', end=' ')

print()

def solve\_queen(size):

board = QueenChessBoard(size)

number\_of\_solutions = 0

row = 0

column = 0

while True:

while column < size:

if board.is\_this\_column\_safe\_in\_next\_row(column):

board.place\_in\_next\_row(column)

row += 1

column = 0

break

else:

column += 1

if (column == size or row == size):

if row == size:

board.display()

print()

number\_of\_solutions += 1

board.remove\_in\_current\_row()

row -= 1

try:

prev\_column = board.remove\_in\_current\_row()

except IndexError:

break

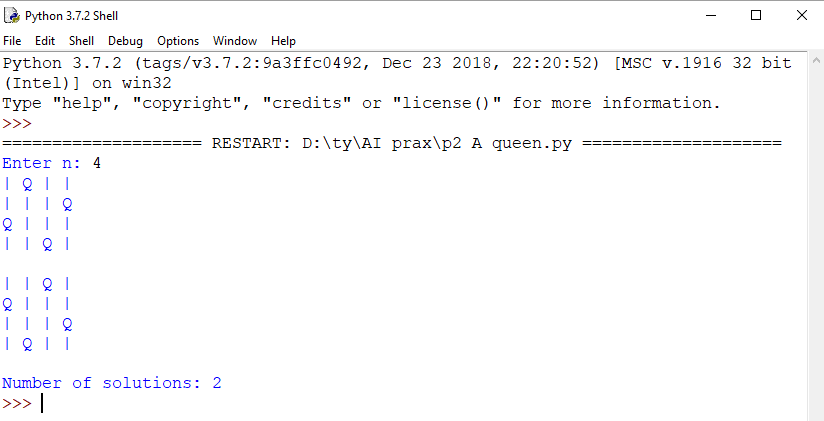
row -= 1

column = 1 + prev\_column

print('Number of solutions:', number\_of\_solutions)

n = int(input('Enter n: '))

solve\_queen(n)



**4] Tower Of Hanoi**

def moveTower(height,fromPole, toPole, withPole):

if height >= 1:

moveTower(height-1,fromPole,withPole,toPole)

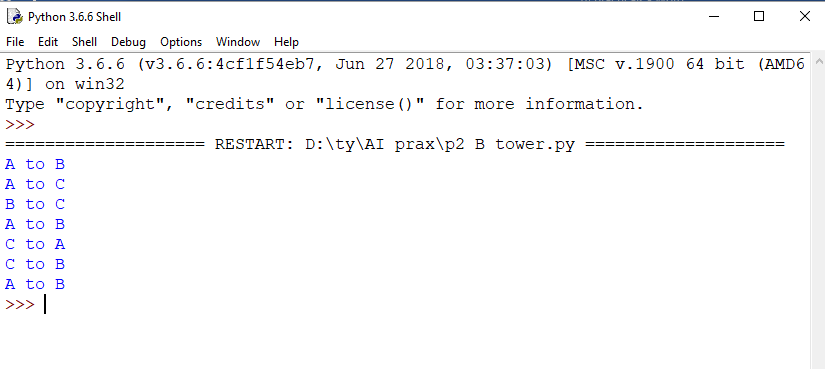
moveDisk(fromPole,toPole)

moveTower(height-1,withPole,toPole,fromPole)

def moveDisk(fp,tp):

print(fp,"to",tp)

moveTower(3,"A","B","C")



**5 ] AlphaBeta**

tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]

root = 0

pruned = 0

def children(branch, depth, alpha, beta):

global tree

global root

global pruned

i = 0

for child in branch:

if type(child) is list:

(nalpha, nbeta) = children(child, depth + 1, alpha, beta)

if depth % 2 == 1:

beta = nalpha if nalpha < beta else beta

else:

alpha = nbeta if nbeta > alpha else alpha

branch[i] = alpha if depth % 2 == 0 else beta

i += 1

else:

if depth % 2 == 0 and alpha < child:

alpha = child

if depth % 2 == 1 and beta > child:

beta = child

if alpha >= beta:

pruned += 1

break

if depth == root:

tree = alpha if root == 0 else beta

return (alpha, beta)

def alphabeta(in\_tree=tree, start=root, upper=-15, lower=15):

global tree

global pruned

global root

(alpha, beta) = children(tree, start, upper, lower)

if \_\_name\_\_ == "\_\_main\_\_":

print ("(alpha, beta): ", alpha, beta)

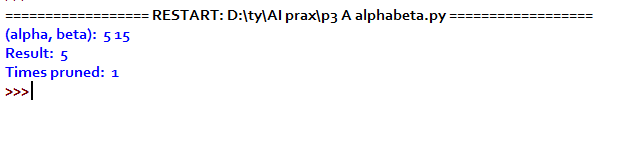
print ("Result: ", tree)

print ("Times pruned: ", pruned)

return (alpha, beta, tree, pruned)

if \_\_name\_\_ == "\_\_main\_\_":

alphabeta(None)



**6] Hill Climbing**

import math

increment = 0.1

startingPoint = [1, 1]

point1 = [1,4 ]

point2 = [3,2]

point3 = [4,0]

point4 = [0,1]

def distance(x1, y1, x2, y2):

dist = math.pow(x2-x1, 2) + math.pow(y2-y1, 2)

return dist

def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):

d1 = distance(x1, y1, px1, py1)

d2 = distance(x1, y1, px2, py2)

d3 = distance(x1, y1, px3, py3)

d4 = distance(x1, y1, px4, py4)

return d1 + d2 + d3 + d4

def newDistance(x1, y1, point1, point2, point3, point4):

d1 = [x1, y1]

d1temp = sumOfDistances(x1, y1, point1[0],point1[1], point2[0],point2[1],

point3[0],point3[1], point4[0],point4[1] )

d1.append(d1temp)

return d1

minDistance = sumOfDistances(startingPoint[0], startingPoint[1], point1[0],point1[1], point2[0],point2[1],

point3[0],point3[1], point4[0],point4[1] )

flag = True

def newPoints(minimum, d1, d2, d3, d4):

if d1[2] == minimum:

return [d1[0], d1[1]]

elif d2[2] == minimum:

return [d2[0], d2[1]]

elif d3[2] == minimum:

return [d3[0], d3[1]]

elif d4[2] == minimum:

return [d4[0], d4[1]]

i = 1

while flag:

d1 = newDistance(startingPoint[0]+increment, startingPoint[1], point1, point2, point3, point4)

d2 = newDistance(startingPoint[0]-increment, startingPoint[1], point1, point2, point3, point4)

d3 = newDistance(startingPoint[0], startingPoint[1]+increment, point1, point2, point3, point4)

d4 = newDistance(startingPoint[0], startingPoint[1]-increment, point1, point2, point3, point4)

print (i,' ', round(startingPoint[0], 2), round(startingPoint[1], 2))

minimum = min(d1[2], d2[2], d3[2], d4[2])

if minimum < minDistance:

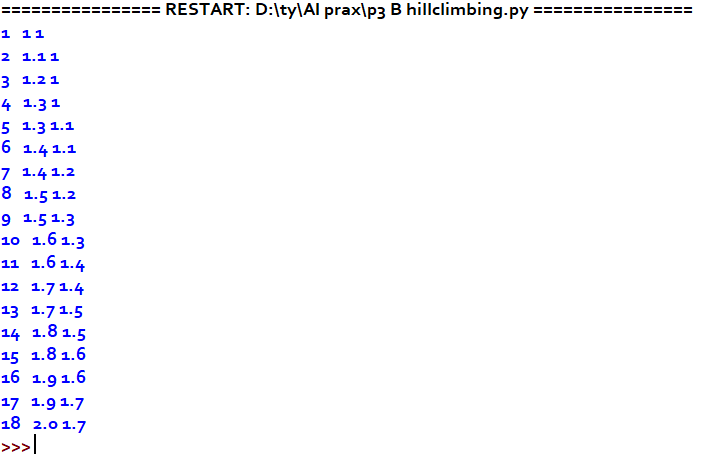
startingPoint = newPoints(minimum, d1, d2, d3, d4)

minDistance = minimum

i+=1

else:

flag = False



**7] Water jug Problem**

capacity = (12,8,5)

# Maximum capacities of 3 jugs ->x,y,z

x = capacity[0]

y = capacity[1]

z = capacity[2]

# to mark visited states

memory = {}

# store solution path

ans = []

def get\_all\_states(state):

# Let the 3 jugs be called a,b,c

a = state[0]

b = state[1]

c = state[2]

if(a==6 and b==6):

ans.append(state)

return True

# if current state is already visited earlier

if((a,b,c) in memory):

return False

memory[(a,b,c)] = 1

#empty jug a

if(a>0):

#empty a into b

if(a+b<=y):

if( get\_all\_states((0,a+b,c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a-(y-b), y, c)) ):

ans.append(state)

return True

#empty a into c

if(a+c<=z):

if( get\_all\_states((0,b,a+c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a-(z-c), b, z)) ):

ans.append(state)

return True

#empty jug b

if(b>0):

#empty b into a

if(a+b<=x):

if( get\_all\_states((a+b, 0, c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((x, b-(x-a), c)) ):

ans.append(state)

return True

#empty b into c

if(b+c<=z):

if( get\_all\_states((a, 0, b+c)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a, b-(z-c), z)) ):

ans.append(state)

return True

if(c>0):

#empty c into a

if(a+c<=x):

if( get\_all\_states((a+c, b, 0)) ):

ans.append(state)

return True

else:

if( get\_all\_states((x, b, c-(x-a))) ):

ans.append(state)

return True

#empty c into b

if(b+c<=y):

if( get\_all\_states((a, b+c, 0)) ):

ans.append(state)

return True

else:

if( get\_all\_states((a, y, c-(y-b))) ):

ans.append(state)

return True

return False

initial\_state = (12,0,0)

print("Starting work...\n")

get\_all\_states(initial\_state)

ans.reverse()

for i in ans:

print(i)



**8] Tic-Tac-Toe Problem**

import os

import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']

player = 1

########win Flags##########

Win = 1

Draw = -1

Running = 0

Stop = 1

###########################

Game = Running

Mark = 'X'

def DrawBoard():

print(" %c | %c | %c " % (board[1],board[2],board[3]))

# print("\_\_\_ |\_\_\_ |\_\_\_")

print(" %c | %c | %c " % (board[4],board[5],board[6]))

# print("\_\_\_ |\_\_\_ |\_\_\_")

print(" %c | %c | %c " % (board[7],board[8],board[9]))

# print(" | | ")

#This Fun ction Checks position is empty or not

def CheckPosition(x):

if(board[x] == ' '):

return True

else:

return False

#This Function Checks player has won or not

def CheckWin():

global Game

#Horizontal winning condition

if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):

Game = Win

elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):

Game = Win

elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):

Game = Win

#Vertical Winning Condition

elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):

Game = Win

elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):

Game = Win

elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):

Game=Win

#Diagonal Winning Condition

elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):

Game = Win

elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):

Game=Win

#Match Tie or Draw Condition

elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):

Game=Draw

else:

Game=Running

print("Tic-Tac-Toe Game")

print("Player 1 [X] --- Player 2 [O]\n")

print()

print("Please Wait...")

time.sleep(1)

while(Game == Running):

os.system('cls')

DrawBoard()

if(player % 2 != 0):

print("Player 1's chance")

Mark = 'X'

else:

print("Player 2's chance")

Mark = 'O'

choice = int(input("Enter the position between [1-9] where you want to mark : "))

if(CheckPosition(choice)):

board[choice] = Mark

player+=1

CheckWin()

os.system('cls')

DrawBoard()

if(Game==Draw):

print("Game Draw")

elif(Game==Win):

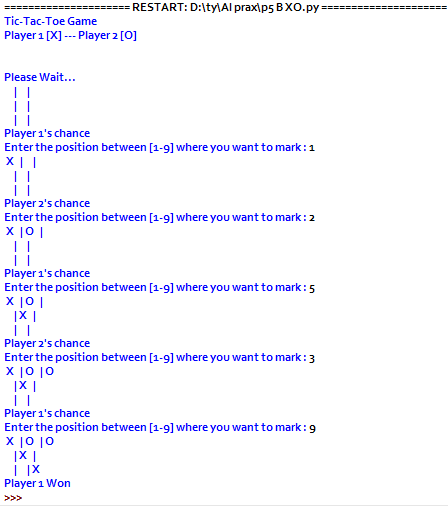
player-=1

if(player%2!=0):

print("Player 1 Won")

else:

print("Player 2 Won")

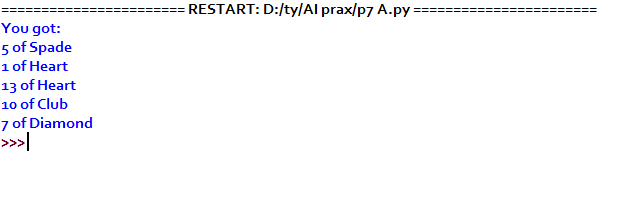


**9] shuffle Deck of cards**  
import itertools, random

# make a deck of cards  
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards  
random.shuffle(deck)

# draw five cards  
print("You got:")  
for i in range(5):  
   print(deck[i][0], "of", deck[i][1])



**10] Travelling Salesman problem in C lang**

#include<stdio.h>

int ary[10][10],completed[10],n,cost=0;

void takeInput()

{

int i,j;

printf("Enter the number of villages: ");

scanf("%d",&n);

printf("\nEnter the Cost Matrix\n");

for(i=0;i < n;i++)

{

printf("\nEnter Elements of Row: %d\n",i+1);

for( j=0;j < n;j++)

scanf("%d",&ary[i][j]);

completed[i]=0;

}

printf("\n\nThe cost list is:");

for( i=0;i < n;i++)

{

printf("\n");

for(j=0;j < n;j++)

printf("\t%d",ary[i][j]);

}

}

void mincost(int city)

{

int i,ncity;

completed[city]=1;

printf("%d--->",city+1);

ncity=least(city);

if(ncity==999)

{

ncity=0;

printf("%d",ncity+1);

cost+=ary[city][ncity];

return;

}

mincost(ncity);

}

int least(int c)

{

int i,nc=999;

int min=999,kmin;

for(i=0;i < n;i++)

{

if((ary[c][i]!=0)&&(completed[i]==0))

if(ary[c][i]+ary[i][c] < min)

{

min=ary[i][0]+ary[c][i];

kmin=ary[c][i];

nc=i;

}

}

if(min!=999)

cost+=kmin;

return nc;

}

int main()

{

takeInput();

printf("\n\nThe Path is:\n");

mincost(0); //passing 0 because starting vertex

printf("\n\nMinimum cost is %d\n ",cost);

return 0;

}

output:

Output

Enter the number of villages: 4

Enter the Cost Matrix

Enter Elements of Row: 1

0 4 1 3

Enter Elements of Row: 2

4 0 2 1

Enter Elements of Row: 3

1 2 0 5

Enter Elements of Row: 4

3 1 5 0

The cost list is:

0 4 1 3

4 0 2 1

1 2 0 5

3 1 5 0

The Path is:

1—>3—>2—>4—>1

Minimum cost is 7

**11] Associative law**

myvar=5\*(5\*6)  
myvar1=(5\*5)+6  
myvar=myvar1  
print(myvar and myvar1)  
print(bool(myvar and myvar1))  
output:

RESTART: C:/Users/Akansha/AppData/Local/Programs/Python/Python37-32/asociative.py  
31  
True

**12] Distributive law**

myvar=(5)\*(6+2-4)  
myvar1=(5)\*6+(5)\*2-(5)+4  
print(myvar and myvar1)  
print(bool(myvar and myvar1))  
output:

 RESTART: C:/Users/Akansha/AppData/Local/Programs/Python/Python37-32/distrib.py  
39  
True

13] constrain satisfaction  
  
  
from \_\_future\_\_ import print\_function  
   
from simpleai.search import CspProblem, backtrack, min\_conflicts, MOST\_CONSTRAINED\_VARIABLE, HIGHEST\_DEGREE\_VARIABLE, LEAST\_CONSTRAINING\_VALUE  
   
variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')  
   
domains = dict((v, ['red', 'green', 'blue'])for v in variables)  
  
   
def const\_different(variables, values):  
    return values[0] != values[1]  
# expect the value of the neighbors to be different  
   
constraints = [     (('WA', 'NT'), const\_different),  
                    (('WA', 'SA'), const\_different),  
                    (('SA', 'NT'), const\_different),  
                    (('SA', 'Q'), const\_different),  
                    (('NT', 'Q'), const\_different),  
                    (('SA', 'NSW'), const\_different),  
                    (('Q', 'NSW'), const\_different),  
                    (('SA', 'V'), const\_different),  
                    (('NSW', 'V'), const\_different),  
                    ]  
   
my\_problem = CspProblem(variables, domains, constraints)  
   
print(backtrack(my\_problem))  
print(backtrack(my\_problem, variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE))  
print(backtrack(my\_problem, variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE))  
print(backtrack(my\_problem, value\_heuristic=LEAST\_CONSTRAINING\_VALUE))  
print(backtrack(my\_problem, variable\_heuristic=MOST\_CONSTRAINED\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE))  
print(backtrack(my\_problem, variable\_heuristic=HIGHEST\_DEGREE\_VARIABLE, value\_heuristic=LEAST\_CONSTRAINING\_VALUE))  
print(min\_conflicts(my\_problem))

14] Missionaries Cannibal

import math  
class State():  
     def \_\_init\_\_(self, cannibalLeft, missionaryLeft, boat, cannibalRight, missionaryRight):  
        self.cannibalLeft = cannibalLeft  
        self.missionaryLeft = missionaryLeft  
        self.boat = boat  
        self.cannibalRight = cannibalRight  
        self.missionaryRight = missionaryRight  
        self.parent = None  
   
     def is\_goal(self):  
         if self.cannibalLeft == 0 and self.missionaryLeft == 0:  
            return True  
         else:  
             return False  
   
     def is\_valid(self):  
         if self.missionaryLeft >= 0 and self.missionaryRight >= 0 \  
             and self.cannibalLeft >= 0 and self.cannibalRight >= 0 \  
             and (self.missionaryLeft == 0 or self.missionaryLeft >= self.cannibalLeft) \  
             and (self.missionaryRight == 0 or self.missionaryRight >= self.cannibalRight):  
                 return True  
         else:  
                 return False  
   
     def \_\_eq\_\_(self, other):  
         return self.cannibalLeft == other.cannibalLeft and self.missionaryLeft == other.missionaryLeft \  
            and self.boat == other.boat and self.cannibalRight == other.cannibalRight \  
            and self.missionaryRight == other.missionaryRight  
   
     def \_\_hash\_\_(self):  
         return hash((self.cannibalLeft, self.missionaryLeft, self.boat, self.cannibalRight, self.missionaryRight))  
     
   
def successors(cur\_state):  
    children = [];  
    if cur\_state.boat == 'left':  
        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 2, 'right',  
  
                        cur\_state.cannibalRight, cur\_state.missionaryRight + 2)  
        ## Two missionaries cross left to right.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft - 2, cur\_state.missionaryLeft, 'right',  
                          cur\_state.cannibalRight + 2, cur\_state.missionaryRight)  
        ## Two cannibals cross left to right.  
        if new\_state.is\_valid():  
                new\_state.parent = cur\_state  
                children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft - 1, 'right',cur\_state.cannibalRight + 1, cur\_state.missionaryRight + 1)  
        ## One missionary and one cannibal cross left to right.  
        if new\_state.is\_valid():  
                new\_state.parent = cur\_state  
                children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft - 1, 'right',  
                                  cur\_state.cannibalRight, cur\_state.missionaryRight + 1)  
                 
        ## One missionary crosses left to right.  
        if new\_state.is\_valid():  
                    new\_state.parent = cur\_state  
                    children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft - 1, cur\_state.missionaryLeft, 'right',  
                                      cur\_state.cannibalRight + 1, cur\_state.missionaryRight) \  
        ## One cannibal crosses left to right.  
        if new\_state.is\_valid():  
                        new\_state.parent = cur\_state  
                        children.append(new\_state)  
    else:  
        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 2, 'left',  
                                      cur\_state.cannibalRight, cur\_state.missionaryRight - 2)  
        ## Two missionaries cross right to left.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft + 2, cur\_state.missionaryLeft, 'left',  
                                          cur\_state.cannibalRight - 2, cur\_state.missionaryRight)  
        ## Two cannibals cross right to left.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft + 1, 'left',  
                          cur\_state.cannibalRight - 1, cur\_state.missionaryRight - 1)  
        ## One missionary and one cannibal cross right to left.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft, cur\_state.missionaryLeft + 1, 'left',  
                          cur\_state.cannibalRight, cur\_state.missionaryRight - 1)  
        ## One missionary crosses right to left.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
        new\_state = State(cur\_state.cannibalLeft + 1, cur\_state.missionaryLeft, 'left',  
                          cur\_state.cannibalRight - 1, cur\_state.missionaryRight)  
        ## One cannibal crosses right to left.  
        if new\_state.is\_valid():  
            new\_state.parent = cur\_state  
            children.append(new\_state)  
    return children  
def breadth\_first\_search():  
     initial\_state = State(3,3,'left',0,0)  
     if initial\_state.is\_goal():  
         return initial\_state  
     frontier = list()  
     explored = set()  
     frontier.append(initial\_state)  
     while frontier:  
            state = frontier.pop(0)  
            if state.is\_goal():  
                return state  
            explored.add(state)  
            children = successors(state)  
            for child in children:  
                if (child not in explored) or (child not in frontier):  
                    frontier.append(child)  
     return None  
def print\_solution(solution):  
     path = []  
     path.append(solution)  
     parent = solution.parent  
     while parent:  
             path.append(parent)  
             parent = parent.parent  
   
     for t in range(len(path)):  
             state = path[len(path) - t - 1]  
             print ("(" + str(state.cannibalLeft) + "," + str(state.missionaryLeft) \  
                + "," + state.boat + "," + str(state.cannibalRight) + "," + \  
                str(state.missionaryRight) + ")")  
def main():  
      solution = breadth\_first\_search()  
      print ("Missionaries and Cannibals solution:")  
      print ("(cannibalLeft,missionaryLeft,boat,cannibalRight,missionaryRight)")  
      print\_solution(solution)  
   
# if called from the command line, call main()  
if \_\_name\_\_ == "\_\_main\_\_":  
        main()

15] A\* algorithm  
  
from simpleai.search import SearchProblem, astar  
   
GOAL = 'HELLO WORLD'  
class HelloProblem(SearchProblem):  
    def actions(self, state):  
        if len(state) < len(GOAL):  
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')  
        else:  
            return []  
   
    def result(self, state, action):  
        return state + action  
    def is\_goal(self, state):  
         return state == GOAL  
   
    def heuristic(self, state):  
        # how far are we from the goal?  
        wrong = sum([1 if state[i] != GOAL[i] else 0  
                     for i in range(len(state))])  
        missing = len(GOAL) - len(state)  
        return wrong + missing  
   
problem = HelloProblem(initial\_state='')  
result = astar(problem)  
print(result.state)  
print(result.path())