**TIC TAC TOE APPROACH2**

|  |
| --- |
| poss={ |
|  | '1':[('2','3'),('4','7'),('5','9')], |
|  | '2':[('1','3'),('5','8')], |
|  | '3':[('1','2'),('6','9'),('5','7')], |
|  | '4':[('1','7'),('5','6')], |
|  | '5':[('4','6'),('2','8'),('1','9'),('3','7')], |
|  | '6':[('3','9'),('4','5')], |
|  | '7':[('1','4'),('8','9'),('5','3')], |
|  | '8':[('7','9'),('5','2')], |
|  | '9':[('6','3'),('7','8'),('1','5')] |
|  | } |
|  |  |
|  | theBoard = {'7':'-','8':'-','9':'-', |
|  | '4':'-','5':'-','6':'-', |
|  | '1':'-','2':'-','3':'-'} |
|  | board\_Keys = [] |
|  | for key in theBoard: |
|  | board\_Keys.append(key) |
|  |  |
|  | def printBoard(board): |
|  | print('-------------') |
|  | print('| '+board['7']+' | '+board['8']+' | '+board['9']+' |') |
|  | print('-------------') |
|  | print('| '+board['4']+' | '+board['5']+' | '+board['6']+' |') |
|  | print('-------------') |
|  | print('| '+board['1']+' | '+board['2']+' | '+board['3']+' |') |
|  | print('-------------') |
|  | printBoard(theBoard) |
|  |  |
|  | def win(board,turn): |
|  | for i in (1,4,7): |
|  | a,b,c=str(i),str(i+1),str(i+2) #row wise win. |
|  | if board[a]==board[b]==board[c]==turn:return True |
|  | for i in (1,2,3): |
|  | a,b,c=str(i),str(i+3),str(i+6) #column wise win. |
|  | if board[a]==board[b]==board[c]==turn:return True |
|  | if board['1']==board['5']==board['9']==turn:return True # 1,5,9 diagonal win. |
|  | if board['3']==board['5']==board['7']==turn:return True # 3,5,7 diagonal win. |
|  | else:return False |
|  |  |
|  |  |
|  | def poswin(board, turn): |
|  | for i in (1,4,7): |
|  | a,b,c=str(i),str(i+1),str(i+2) |
|  | if board[a]==board[b]==turn or board[b]==board[c]==turn or board[c]==board[a]==turn: #row wise. |
|  | if board[a]=='-':return a |
|  | elif board[b]=='-':return b |
|  | elif board[c]=='-':return c |
|  |  |
|  | for i in (1,2,3): |
|  | a,b,c=str(i),str(i+3),str(i+6) |
|  | if board[a]==board[b]==turn or board[b]==board[c]==turn or board[c]==board[a]==turn: #column wise. |
|  | if board[a]=='-':return a |
|  | elif board[b]=='-':return b |
|  | elif board[c]=='-':return c |
|  |  |
|  | a,b,c='1','5','9' |
|  | if board[a]==board[b]==turn or board[b]==board[c]==turn or board[c]==board[a]==turn: # 1,5,9 diagonal. |
|  | if board[a]=='-':return a |
|  | elif board[b]=='-':return b |
|  | elif board[c]=='-':return c |
|  |  |
|  | a,b,c='3','5','7' |
|  | if board[a]==board[b]==turn or board[b]==board[c]==turn or board[c]==board[a]==turn: # 3,5,7 diagonal. |
|  | if board[a]=='-':return a |
|  | elif board[b]=='-':return b |
|  | elif board[c]=='-':return c |
|  |  |
|  | return '0' |
|  |  |
|  | def game(): |
|  | count=0 |
|  | while True: |
|  | if count==9: |
|  | print("!! Game Over !!") |
|  | print("It's a tie.") |
|  | break |
|  | move=input("It's your turn. place at ? ") |
|  | if theBoard[move]=='-': |
|  | theBoard[move]='X' |
|  | count+=1 |
|  | else: |
|  | print("The place is already filled.\nplace at ?") |
|  | continue |
|  | printBoard(theBoard) |
|  | if count>4 and win(theBoard,'X'): |
|  | print("!! Game Over !!") |
|  | print("You have Won.") |
|  | break |
|  | print("A.I's turn.") |
|  | if move!='5' and theBoard['5']=='-':theBoard['5']='O' |
|  | else: |
|  | pos = poswin(theBoard, 'O') |
|  | if pos!='0': |
|  | theBoard[pos]='O' |
|  | else: |
|  | pos=poswin(theBoard, 'X') |
|  | if pos=='0': |
|  | for p in poss[move]: |
|  | if theBoard[p[0]] == '-': |
|  | theBoard[p[0]] = 'O' |
|  | break |
|  | elif theBoard[p[1]] == '-': |
|  | theBoard[p[1]] = 'O' |
|  | break |
|  | else:theBoard[pos]='O' |
|  | count+=1 |
|  | printBoard(theBoard) |
|  | if count>4 and win(theBoard,'O'): |
|  | print("!! Game Over !!") |
|  | print("A.I won.") |
|  | break |
|  | game() |

**TIC TAC TOE APPROACH3**

|  |
| --- |
| import itertools as it |
|  | poss={ |
|  | 1:[(2,3),(4,7),(5,9)], |
|  | 2:[(1,3),(5,8)], |
|  | 3:[(1,2),(6,9),(5,7)], |
|  | 4:[(1,7),(5,6)], |
|  | 5:[(4,6),(2,8),(1,9),(3,7)], |
|  | 6:[(3,9),(4,5)], |
|  | 7:[(1,4),(8,9),(5,3)], |
|  | 8:[(7,9),(5,2)], |
|  | 9:[(6,3),(7,8),(1,5)] |
|  | } |
|  |  |
|  | ms={7:8, 8:1, 9:6, |
|  | 4:3, 5:5, 6:7, |
|  | 1:4, 2:9, 3:2} |
|  |  |
|  | theBoard = {7:'-',8:'-',9:'-', |
|  | 4:'-',5:'-',6:'-', |
|  | 1:'-',2:'-',3:'-'} |
|  |  |
|  | board\_Keys = [] |
|  | for key in theBoard: |
|  | board\_Keys.append(key) |
|  |  |
|  | def printBoard(board): |
|  | print('-------------') |
|  | print('| '+board[7]+' | '+board[8]+' | '+board[9]+' |') |
|  | print('-------------') |
|  | print('| '+board[4]+' | '+board[5]+' | '+board[6]+' |') |
|  | print('-------------') |
|  | print('| '+board[1]+' | '+board[2]+' | '+board[3]+' |') |
|  | print('-------------') |
|  | printBoard(theBoard) |
|  |  |
|  | def win(board,turn): |
|  | for i in (1,4,7): |
|  | a,b,c=i,i+1,i+2 #row wise win. |
|  | if board[a]==board[b]==board[c]==turn:return True |
|  | for i in (1,2,3): |
|  | a,b,c=i,i+3,i+6 #column wise win. |
|  | if board[a]==board[b]==board[c]==turn:return True |
|  | if board[1]==board[5]==board[9]==turn:return True # 1,5,9 diagonal win. |
|  | if board[3]==board[5]==board[7]==turn:return True # 3,5,7 diagonal win. |
|  | else:return False |
|  |  |
|  | def poswin(board, temp): |
|  | keys=list(ms.keys()) |
|  | vals=list(ms.values()) |
|  | for i in it.combinations(temp,2): |
|  | pos=15-sum(i) |
|  | if pos<=9 and pos>=1: |
|  | val=keys[vals.index(pos)] |
|  | if board[val]=='-':return val |
|  | return 0 |
|  |  |
|  | def game(): |
|  | count=0 |
|  | hu,ai=[],[] |
|  | while True: |
|  | if count==9: |
|  | print("!! Game Over !!") |
|  | print("It's a tie.") |
|  | break |
|  | move=int(input("It's your turn. place at ? ")) |
|  | if theBoard[move]=='-': |
|  | theBoard[move]='X' |
|  | hu.append(ms[move]) |
|  | count+=1 |
|  | else: |
|  | print("The place is already filled.\nplace at ?") |
|  | continue |
|  | printBoard(theBoard) |
|  | if count>4 and win(theBoard, 'X'): |
|  | print("!! Game Over !!") |
|  | print("You have Won.") |
|  | break |
|  | print("A.I's turn.") |
|  | if move!=5 and theBoard[5]=='-': |
|  | theBoard[5]='O' |
|  | ai.append(ms[5]) |
|  | else: |
|  | pos = poswin(theBoard, ai) |
|  | if pos!=0: |
|  | theBoard[pos]='O' |
|  | ai.append(ms[pos]) |
|  | else: |
|  | pos=poswin(theBoard, hu) |
|  | if pos==0: |
|  | for p in poss[move]: |
|  | if theBoard[p[0]] == '-': |
|  | theBoard[p[0]] = 'O' |
|  | ai.append(ms[p[0]]) |
|  | break |
|  | elif theBoard[p[1]] == '-': |
|  | theBoard[p[1]] = 'O' |
|  | ai.append(ms[p[1]]) |
|  | break |
|  | else: |
|  | theBoard[pos]='O' |
|  | ai.append(ms[pos]) |
|  | count+=1 |
|  | printBoard(theBoard) |
|  | if count>4 and win(theBoard, 'O'): |
|  | print("!! Game Over !!") |
|  | print("A.I won.") |
|  | break |
|  | game() |

**WATERJUG BFS & DFS**

class WaterJug:

def \_\_init\_\_(self,bjmax,sjmax,target):

self.bjmax=bjmax

self.sjmax=sjmax

self.goal=target

def bfs(self):

print('BFS Approach : ')

opened,closed=[],[]

opened.append((0,0))

while opened:

p=opened.pop(0)

closed.append(p)

# Goal State

if self.goal in p:

print(p,sep='\t')

print('Goal State is attained');return

# Rule - 1 fill Small Jug

if p[1]==0 and (p[0],self.sjmax) not in closed+opened:opened.append((p[0],self.sjmax))

# Rule - 2 empty Big Jug

if p[0]==self.bjmax and (0,p[1]) not in closed+opened: opened.append((0,p[1]))

# Rule - 3 empty Small Jug to Big Jug

if p[0]+p[1]<=self.bjmax and (p[0]+p[1],0) not in closed+opened: opened.append((p[0]+p[1],0))

# Rule - 4 transfer Small Jug to Big Jug

if p[0]+p[1]>self.bjmax:

temp=self.bjmax-p[0]

temp=(p[0]+temp,p[1]-temp)

if temp not in opened+closed: opened.append(temp)

# Rule - 5 fill Big Jug

if p[0]==0 and (self.bjmax,p[1]) not in closed+opened:

opened.append((self.bjmax,p[1]))

# Rule - 6 empty Small Jug

if p[1]==self.sjmax and (p[0],0) not in closed+opened:

opened.append((p[0],0))

# Rule - 7 Empty Big Jug to Small Jug

if p[0]-p[1]>=0 and p[0]-p[1]<=self.sjmax:

if (p[0]-p[1],p[1]) not in closed+opened:opened.append((p[0]-p[1],p[1]))

print(p,sep='\t')

print('Goal State not Possible')

def dfs(self):

print('DFS Approach : ')

opened,closed=[],[]

opened.append((0,0))

while opened:

p=opened.pop(0)

li = []

closed.append(p)

# Goal State

if self.goal in p:

print(p,sep='\t')

print('Goal State is attained');return

# Rule - 1 fill Small Jug

if p[1]==0 and (p[0],self.sjmax) not in closed+opened:li.append((p[0],self.sjmax))

# Rule - 2 empty Big Jug

if p[0]==self.bjmax and (0,p[1]) not in closed+opened: li.append((0,p[1]))

# Rule - 3 empty Small Jug to Big Jug

if p[0]+p[1]<=self.bjmax and (p[0]+p[1],0) not in closed+opened: li.append((p[0]+p[1],0))

# Rule - 4 transfer Small Jug to Big Jug

if p[0]+p[1]>self.bjmax:

temp=self.bjmax-p[0]

temp=(p[0]+temp,p[1]-temp)

if temp not in opened+closed: li.append(temp)

# Rule - 5 fill Big Jug

if p[0]==0 and (self.bjmax,p[1]) not in closed+opened:

li.append((self.bjmax,p[1]))

# Rule - 6 empty Small Jug

if p[1]==self.sjmax and (p[0],0) not in closed+opened:

li.append((p[0],0))

# Rule - 7 Empty Big Jug to Small Jug

if p[0]-p[1]>=0 and p[0]-p[1]<=self.sjmax:

if (p[0]-p[1],p[1]) not in closed+opened:opened.append((p[0]-p[1],p[1]))

opened=li+opened

print(p,sep='\t')

print('Goal State not Possible')

bjmax,sjmax,target=map(int,input('Enter capacities of jugs and target : ').split())

w=WaterJug(bjmax, sjmax, target)

w.bfs()

w.dfs()

**BFS, DFS GENERAL**

def BFS(graph,start,goal):

open,closed=[],[]

open.append(start)

closed.append(start)

while(open):

i=open.pop(0)

print(i,end=" ")

if(i==goal):

print("Goal state found")

break

for j in graph[i]:

if j not in closed:

open.append(j)

closed.append(j)

def DFS(graph,start,goal):

open,closed=[],[]

closed.append(start)

open.append(start)

while(open):

i=open.pop()

print(i,end=" ")

if(i==goal):

print("Goal state found")

break

succ=graph[i]

for j in succ[::-1]:

if j not in closed:

closed.append(j)

open.append(j)

n=int(input("enter number of nodes"))

graph={}

for i in range(n):

k=[]

print("enter nodes linked to ",i)

for j in input().split():

k.append(int(j))

graph[i]=k

print(graph)

start=int(input("enter start node "))

goal=int(input("enter goal node "))

BFS(graph,start,goal)

DFS(graph,start,goal)

**DFID**

class ExhaustiveSearch:  
    def \_\_init\_\_(self,graph,start,goal):  
        self.graph=graph  
        self.start=start  
        self.goal=goal

  lst=[]  
    def dSearch(self,start,goal,depth):  
        global lst  
        lst.append(start)  
        if start==goal:  
            return True  
        if depth<=0:  
            return False  
        for i in self.graph[start]:  
            if self.dSearch(i,goal,depth-1):  
                return True  
        return False  
    def dfid(self):  
        global lst  
        for i in range(maxLevel):  
            lst=[]  
            x=self.dSearch(self.start,self.goal,i)  
            print('Depth '+str(i)+' : ',lst)  
            if x:  
                print('Goal node found with DFID with depth '+str(i))  
                break  
                     
         
if \_\_name\_\_=="\_\_main\_\_":  
    n=int(input('Enter the number of nodes: '))  
    graph={}  
    for i in range(n):  
        print('Enter nodes linked to '+str(i)+' :',end=' ')  
        lst=[int(x) for x in input().split()]  
        graph[i]=lst  
    start=int(input('Enter the start node: '))  
    goal=int(input('Enter goal node: '))  
    maxLevel=int(input('Enter the maximum depth of the tree: '))  
    alg=ExhaustiveSearch(graph,start,goal)  
     
    alg.dfid()

**BIDRECTIONAL**

class Graph:

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

self.g = defaultdict(list)

self.c1,self.c2 = [],[]

def addedge(self,u,v):

self.g[u].append(v)

self.g[v].append(u)

def bidirectional(self,start,goal):

o1,o2 = [],[]

o1.append(start)

o2.append(goal)

while o1 and o2:

t1 = o1.pop(0)

self.c1.append(t1)

t2 = o2.pop(0)

self.c2.append(t2)

for i in self.g[t1]:

if i not in self.c1:

o1.append(i)

for i in self.g[t2]:

if i not in self.c2:

o2.append(i)

for i in o1:

if i in o2:print(o1, " ", self.c1, " ", o2, " ", self.c2); print('Goal Found'); return

print(o1, " ", self.c1, " ", o2, " ", self.c2)

print("Not found")

g = Graph()

n = int(input("No.of edges...."))

for i in range(n):

u,v = input().split()

g.addedge(u,v)

start, goal = input().split()

g.bidirectional(start,goal)

O/P:

No.of edges....9

a b

a c

a d

b e

b f

c g

c h

d i

d j

a z

['b', 'c', 'd'] ['a'] [] ['z']

Not found

**HILL CLIMBING**

from collections import defaultdict

class Graph:

def \_\_init\_\_(self,n,h):

self.n=n

self.h=h

self.graph=defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

self.graph[v].append(u)

def hillClimb(self,start,goal):

print('Hill Climbing Search : ')

print("open","close",sep='\t\t\t')

opened,closed=[],[]

opened.append(start)

print(opened, closed, sep='\t\t\t')

while opened:

p=opened.pop(0)

closed.insert(0,p)

opened.sort(key=lambda x: self.h[x])

#Goal node

if p==goal:

print(opened,closed,sep='\t\t\t')

print('Goal node found');return

#Successors Generation

for v in self.graph[p]:

if v not in opened and v not in closed:opened.insert(0,v)

print(opened,closed,sep='\t\t\t')

print('Goal node not found')

n=int(input('Enter no.of nodes: '))

h={}

for \_ in range(n):

u,i=input('Enter node and it\'s heuristic: ').split()

h[u]=int(i)

g=Graph(n,h)

m=int(input('Enter no.of edges: '))

for \_ in range(m):

u,v=input('Enter edge nodes: ').split()

g.addEdge(u,v)

start,goal=input('Enter start and goal states: ').split()

g.hillClimb(start,goal)

O/P:

Enter no.of nodes: 12

Enter node and it's heuristic: a 10

Enter node and it's heuristic: b 10

Enter node and it's heuristic: i 8

Enter node and it's heuristic: f 7

Enter node and it's heuristic: d 4

Enter node and it's heuristic: c 2

Enter node and it's heuristic: h 0

Enter node and it's heuristic: k 0

Enter node and it's heuristic: e 5

Enter node and it's heuristic: j 6

Enter node and it's heuristic: g 3

Enter node and it's heuristic: m 0

Enter no.of edges: 11

Enter edge nodes: a b

Enter edge nodes: a i

Enter edge nodes: a f

Enter edge nodes: b d

Enter edge nodes: b c

Enter edge nodes: c h

Enter edge nodes: i k

Enter edge nodes: f e

Enter edge nodes: f g

Enter edge nodes: e j

Enter edge nodes: j m

Enter start and goal states: a k

Hill Climbing Search :

open close

['a'] []

['f', 'i', 'b'] ['a']

['g', 'e', 'i', 'b'] ['f', 'a']

['e', 'i', 'b'] ['g', 'f', 'a']

['j', 'i', 'b'] ['e', 'g', 'f', 'a']

['m', 'i', 'b'] ['j', 'e', 'g', 'f', 'a']

['i', 'b'] ['m', 'j', 'e', 'g', 'f', 'a']

['k', 'b'] ['i', 'm', 'j', 'e', 'g', 'f', 'a']

['b'] ['k', 'i', 'm', 'j', 'e', 'g', 'f', 'a']

Goal node found

**MISSIONARIES AND CANNIBALLS**

**BFS**

m=int(input("no.of missionaries:"))  
c=int(input("no.of cannibals:"))  
init=[[(m,c,1),(0,0,0)]]  
final=[(0,0,0),(m,c,1)]  
opened=init[:]  
closed=[]  
while(opened):  
    print("open=",opened)  
    k=opened.pop(0)  
    closed.append(k)  
    print("closed=",closed)  
    if k==final:  
        print("goal state reached")  
        break  
    if k[0][2]==1:  
        if k[0][0]>=2:  
            if k[0][0]-2>=k[0][1]:  
                if [(k[0][0]-2,k[0][1],0),(k[1][0]+2,k[1][1],1)] not in opened+closed:  
                    opened.append([(k[0][0]-2,k[0][1],0),(k[1][0]+2,k[1][1],1)])  
        if k[0][0]>1 and k[0][1]>1:  
            if k[0][0]-1>=k[0][1]-1 and [(k[0][0]-1,k[0][1]-1,0),(k[1][0]+1,k[1][1]+1,1)] not in opened+closed:  
                opened.append([(k[0][0]-1,k[0][1]-1,0),(k[1][0]+1,k[1][1]+1,1)])  
        if k[0][1]>=2:  
            if k[0][0]>=k[0][1]-2 and [(k[0][0],k[0][1]-2,0),(k[1][0],k[1][1]+2,1)] not in opened+closed:  
                opened.append([(k[0][0],k[0][1]-2,0),(k[1][0],k[1][1]+2,1)])  
        if k[0][0]>=1:  
            if k[0][0]-1>=k[0][1] and [(k[0][0]-1,k[0][1],0),(k[1][0]+1,k[1][1],1)] not in opened+closed:  
                opened.append([(k[0][0]-1,k[0][1],0),(k[1][0]+1,k[1][1],1)])  
        if k[0][1]>=1:  
            if k[0][0]>=k[0][1]-1 and [(k[0][0],k[0][1]-1,0),(k[1][0],k[1][1]+1,1)] not in opened+closed:  
                opened.append([(k[0][0],k[0][1]-1,0),(k[1][0],k[1][1]+1,1)])  
    elif k[0][2]==0:  
        if k[1][0]>=2:  
            if k[0][0]+2>=k[0][1] and [(k[0][0]+2,k[0][1],1),(k[1][0]-2,k[1][1],0)] not in opened+closed:  
                opened.append([(k[0][0]+2,k[0][1],1),(k[1][0]-2,k[1][1],0)])  
        if k[1][0]>1 and k[1][1]>1:  
            if k[0][0]+1>=k[0][1]+1 and [(k[0][0]+1,k[0][1]+1,1),(k[1][0]-1,k[1][1]-1,0)] not in opened+closed:  
                opened.append([(k[0][0]+1,k[0][1]+1,1),(k[1][0]-1,k[1][1]-1,0)])  
        if k[1][1]>=2:  
            if k[0][0]>=k[0][1]+2 and [(k[0][0],k[0][1]+2,1),(k[1][0],k[1][1]-2,0)] not in opened+closed:  
                opened.append([(k[0][0],k[0][1]+2,1),(k[1][0],k[1][1]-2,0)])  
        if k[1][0]>=1:  
            if k[0][0]+1>=k[0][1] and [(k[0][0]+1,k[0][1],1),(k[1][0]-1,k[1][1],0)] not in opened+closed:  
                opened.append([(k[0][0]+1,k[0][1],1),(k[1][0]-1,k[1][1],0)])  
        if k[1][1]>=1:  
            if k[0][0]>=k[0][1]+1 and [(k[0][0],k[0][1]+1,1),(k[1][0],k[1][1]-1,0)] not in opened+closed:  
                opened.append([(k[0][0],k[0][1]+1,1),(k[1][0],k[1][1]-1,0)])  
else:  
    print("goal state cant be reached")

**DFS**

m=int(input("no.of missionaries:"))  
c=int(input("no.of cannibals:"))  
init=[[(m,c,1),(0,0,0)]]  
final=[(0,0,0),(m,c,1)]  
opened=init[:]  
closed=[]  
while(opened):  
    print("open=",opened)  
    k=opened.pop(0)  
    closed.append(k)  
    print("closed=",closed)  
    if k==final:  
        print("goal state reached")  
        break  
    if k[0][2]==1:  
        if k[0][1]>=1:  
            if k[0][0]>=k[0][1]-1 and [(k[0][0],k[0][1]-1,0),(k[1][0],k[1][1]+1,1)] not in opened+closed:  
                opened.insert(0,[(k[0][0],k[0][1]-1,0),(k[1][0],k[1][1]+1,1)])  
        if k[0][0]>=1:  
            if k[0][0]-1>=k[0][1] and [(k[0][0]-1,k[0][1],0),(k[1][0]+1,k[1][1],1)] not in opened+closed:  
                opened.insert(0,[(k[0][0]-1,k[0][1],0),(k[1][0]+1,k[1][1],1)])  
        if k[0][1]>=2:  
            if k[0][0]>=k[0][1]-2 and [(k[0][0],k[0][1]-2,0),(k[1][0],k[1][1]+2,1)] not in opened+closed:  
                opened.insert(0,[(k[0][0],k[0][1]-2,0),(k[1][0],k[1][1]+2,1)])  
        if k[0][0]>=1 and k[0][1]>=1:  
            if k[0][0]-1>=k[0][1]-1 and [(k[0][0]-1,k[0][1]-1,0),(k[1][0]+1,k[1][1]+1,1)] not in opened+closed:  
                opened.insert(0,[(k[0][0]-1,k[0][1]-1,0),(k[1][0]+1,k[1][1]+1,1)])  
        if k[0][0]>=2:  
            if k[0][0]-2>=k[0][1]:  
                if [(k[0][0]-2,k[0][1],0),(k[1][0]+2,k[1][1],1)] not in opened+closed:  
                    opened.insert(0,[(k[0][0]-2,k[0][1],0),(k[1][0]+2,k[1][1],1)])  
    elif k[0][2]==0:  
        if k[1][1]>=1:  
            if k[0][0]>=k[0][1]+1 and [(k[0][0],k[0][1]+1,1),(k[1][0],k[1][1]-1,0)] not in opened+closed:  
                opened.insert(0,[(k[0][0],k[0][1]+1,1),(k[1][0],k[1][1]-1,0)])  
        if k[1][0]>=1:  
            if k[0][0]+1>=k[0][1] and [(k[0][0]+1,k[0][1],1),(k[1][0]-1,k[1][1],0)] not in opened+closed:  
                opened.insert(0,[(k[0][0]+1,k[0][1],1),(k[1][0]-1,k[1][1],0)])  
        if k[1][1]>=2:  
            if k[0][0]>=k[0][1]+2 and [(k[0][0],k[0][1]+2,1),(k[1][0],k[1][1]-2,0)] not in opened+closed:  
                opened.insert(0,[(k[0][0],k[0][1]+2,1),(k[1][0],k[1][1]-2,0)])  
        if k[1][0]>=1 and k[1][1]>=1:  
            if k[0][0]+1>=k[0][1]+1 and [(k[0][0]+1,k[0][1]+1,1),(k[1][0]-1,k[1][1]-1,0)] not in opened+closed:  
                opened.insert(0,[(k[0][0]+1,k[0][1]+1,1),(k[1][0]-1,k[1][1]-1,0)])  
        if k[1][0]>=2:  
            if k[0][0]+2>=k[0][1] and [(k[0][0]+2,k[0][1],1),(k[1][0]-2,k[1][1],0)] not in opened+closed:  
                opened.insert(0,[(k[0][0]+2,k[0][1],1),(k[1][0]-2,k[1][1],0)])  
else:  
    print("goal state cant be reached")