**BFS**

n = int(input("Enter No.of nodes : "))

Graph = list()

# Using Adjacency matrix

print("Enter Adj Matrix : ")

for i in range(n):

row = list(map(int, input().split()))

Graph.append(row)

# Using Ajacency List

# Graph = [[1e9 for i in range(n)] for j in range(n)]

# e = int(input("Enter No.of edges : "))

# for i in range(e):

# a,b,w = map(int, input("Enter the edge points and weight : ").split())

# Graph[a][b] = w

# Graph[b][a] = w #iff bi-directional

# Custom Input

# Graph = [[0,2,0,0,10,0,0],[2,0,4,0,0,0,0],[0,4,0,3,0,0,6],[0,0,3,0,0,0,9],[10,0,0,0,0,7,0],[0,0,0,0,7,0,8],[0,0,6,9,0,8,0]]

for row in Graph:

print(row)

Status = [0 for i in range(n)]

Queue = []

Processed = []

start = int(input("Enter starting node : "))

Status[start] = 1

Queue.append(start)

# key = int(input("Enter goal node : ")) # In case of searching

while len(Queue)!=0:

start = Queue[0]

Queue.remove(start)

if start not in Processed:

Processed.append(start)

# In case of searching

# if start==key:

# print(start)

# break

print(start," ==> ",end="")

for i in range(n):

if Graph[start][i]!=0 and Status[i]==0:

Queue.append(i)

Status[i]=1

**DFS**

n = int(input("Enter No.of nodes : "))

Graph = list()

# Using Adjacency matrix

print("Enter Adj Matrix : ")

for i in range(n):

row = list(map(int, input().split()))

Graph.append(row)

# Using Ajacency List

# Graph = [[1e9 for i in range(n)] for j in range(n)]

# e = int(input("Enter No.of Vertices : "))

# for i in range(e):

# a,b,w = map(int, input("Enter the edge points and weight : ").split())

# Graph[a][b] = w

# Graph[b][a] = w #iff bi-directional

# Custom Input

# Graph = [[0,2,0,0,10,0,0],[2,0,4,0,0,0,0],[0,4,0,3,0,0,6],[0,0,3,0,0,0,9],[10,0,0,0,0,7,0],[0,0,0,0,7,0,8],[0,0,6,9,0,8,0]]

for row in Graph:

print(row)

Status = [0 for i in range(n)]

Stack = []

Processed = []

start = int(input("Enter starting node : "))

Status[start] = 1

Stack.append(start)

# key = int(input("Enter goal node : ")) # In case of searching

while len(Stack)!=0:

start = Stack.pop()

if start not in Processed:

Processed.append(start)

# In case of searching

# if start==key:

# print(start)

# break

print(start," ==> ",end="")

for i in range(n):

if M[start][i]!=0 and Status[i]==0:

Stack.append(i)

Status[i]=1

**BFSH**

v = int(input('Enter No.of Vertices : '))

Vertices = dict()

Graph = dict()

for i in range(v):

a,b = input('Enter vertex name and Heuristic value : ').split()

V[str(a)] = int(b)

G[str(a)] = list()

Vertices = dict(sorted(Vertices.items(), key = lambda V:(V[1], V[0])))

e = int(input('Enter No.of Edges : '))

for i in range(e):

a,b = input('Enter the edge points : ').split()

Graph[str(a)].append(str(b))

Start = str(input('Enter initial vertex : '))

Goal = str(input('Enter goal vertex : '))

Open = dict()

Closed = dict()

Open[S] = V[S]

while len(Open)!=0:

Open = dict(sorted(Open.items(), key = lambda Open:(Open[1], Open[0])))

Key = list(Open.keys())[0]

Closed[Key] = Open[Key]

Open.pop(Key)

if Key==Goal:

break

else:

for i in Graph[Key]:

Open[i] = Vertices[i]

print(\*list(Closed.keys()),sep=" ---> ")

**8 Puzzle**

import copy

# print("Enter initial state : ")

# IS = list()

# for i in range(3):

# row = list(map(int, input().split()))

# IS.append(row)

# print("Enter final state : ")

# FS = list()

# for i in range(3):

# row = list(map(int, input().split()))

# FS.append(row)

FS = [[1,2,3],[4,5,6],[7,8,0]]

IS = [[1,2,3],[5,0,6],[4,7,8]]

# IS = [[1,2,3],[0,4,6],[7,5,8]]

# IS = [[1,2,3],[8,0,4],[7,6,5]]

# FS = [[2,8,1],[0,4,3],[7,6,5]]

Processed = list()

Queue = list()

Queue.append(IS)

c=0

while len(Queue)!=0:

c+=1

P = copy.deepcopy(Queue.pop())

for i in P:

print(i)

print()

if P==FS:

print('Final State : ')

print(P)

break

elif P in Processed:

pass

else:

Processed.insert(0,P)

x=-1

y=-1

for i in P:

x+=1

if 0 in i:

for j in i:

y+=1

if j==0:

break

break

if y!=0:

L = copy.deepcopy(P)

L[x][y-1] = P[x][y]

L[x][y] = P[x][y-1]

Queue.insert(0,L)

if y!=2:

R = copy.deepcopy(P)

R[x][y+1] = P[x][y]

R[x][y] = P[x][y+1]

Queue.insert(0,R)

if x!=0:

U = copy.deepcopy(P)

U[x-1][y] = P[x][y]

U[x][y] = P[x-1][y]

Queue.insert(0,U)

if x!=2:

D = copy.deepcopy(P)

D[x+1][y] = P[x][y]

D[x][y] = P[x+1][y]

Queue.insert(0,D)

print("Total states traversed :",c)

**8 Puzzle Heuiristic**

import copy

def heuristic(CS,FS):

h=0

for i in range(3):

for j in range(3):

if CS[i][j]!=FS[i][j]:

h+=1

return h

# print("Enter initial state : ")

# IS = list()

# for i in range(3):

# row = list(map(int, input().split()))

# IS.append(row)

# print("Enter final state : ")

# FS = list()

# for i in range(3):

# row = list(map(int, input().split()))

# FS.append(row)

FS = [[1,2,3],[4,5,6],[7,8,0]]

IS = [[1,2,3],[5,0,6],[4,7,8]]

#IS = [[1,2,3],[0,4,6],[7,5,8]]

# IS = [[1,2,3],[8,0,4],[7,6,5]]

# FS = [[2,8,1],[0,4,3],[7,6,5]]

Processed = list()

Stack = dict()

for i in range(10):

Stack[i] = list()

Stack[heuristic(IS,FS)].append(IS)

print(Stack)

c=0

while True:

c+=1

for i in range(9):

print(Stack[i])

for i in range(9):

try:

P = copy.deepcopy(Stack[i].pop())

break

except:

pass

for i in P:

print(i)

print()

if P==FS:

print('Final State : ')

print(P)

break

else:

Processed.insert(0,P)

x=-1

y=-1

for i in P:

x+=1

if 0 in i:

for j in i:

y+=1

if j==0:

break

break

if y!=0:

L = copy.deepcopy(P)

L[x][y-1] = P[x][y]

L[x][y] = P[x][y-1]

if L not in Processed:

Stack[heuristic(L,FS)].append(L)

if y!=2:

R = copy.deepcopy(P)

R[x][y+1] = P[x][y]

R[x][y] = P[x][y+1]

if R not in Processed:

Stack[heuristic(R,FS)].append(R)

if x!=0:

U = copy.deepcopy(P)

U[x-1][y] = P[x][y]

U[x][y] = P[x-1][y]

if U not in Processed:

Stack[heuristic(U,FS)].append(U)

if x!=2:

D = copy.deepcopy(P)

D[x+1][y] = P[x][y]

D[x][y] = P[x+1][y]

if D not in Processed:

Stack[heuristic(D,FS)].append(D)

print("Total states traversed :",c)

**Missinary Canibels**

class MissionaryCanibels():

def \_\_init\_\_(self, canibels\_left, missionary\_left, boat, canibels\_right, missionary\_right):

self.canibels\_left = canibels\_left

self.missionary\_left = missionary\_left

self.boat = boat

self.canibels\_right = canibels\_right

self.missionary\_right = missionary\_right

self.parent = None

def isGoal(self):

return self.canibels\_left==0 and self.missionary\_left==0

def isValid(self):

if self.canibels\_left<0 or self.missionary\_left<0 or self.canibels\_right<0 or self.missionary\_right<0:

return False

if self.missionary\_left<self.canibels\_left and self.missionary\_left!=0:

return False

if self.missionary\_right<self.canibels\_right and self.missionary\_right!=0:

return False

return True

def generateChild(S):

Children = []

if S.boat=='Left':

for Case in [[2,0],[0,2],[1,1],[1,0],[0,1]]:

Child = MissionaryCanibels(S.canibels\_left-Case[0], S.missionary\_left-Case[1], 'Right', S.canibels\_right+Case[0], S.missionary\_right+Case[1])

Child.parent = S

if Child.isValid():

Children.append(Child)

else:

for Case in [[1,0],[0,1],[1,1],[2,0],[0,2]]:

Child = MissionaryCanibels(S.canibels\_left+Case[0], S.missionary\_left+Case[1], 'Left', S.canibels\_right-Case[0], S.missionary\_right-Case[1])

Child.parent = S

if Child.isValid():

Children.append(Child)

return Children

IS = MissionaryCanibels(3, 3, 'Left', 0, 0)

Queue = list()

Processed = list()

Queue.append(IS)

while len(Queue)!=0:

CS = Queue[0]

Queue.remove(CS)

if CS.isGoal():

Solution = []

Solution.append(CS)

while CS.parent!=None:

CS = CS.parent

Solution.append(CS)

for i in Solution[::-1]:

print(i.canibels\_left, i.missionary\_left, i.boat, i.canibels\_right, i.missionary\_right)

print()

# break

elif CS in Processed:

pass

else:

Processed.append(CS)

Children = generateChild(CS)

for Child in Children:

if Child not in Queue and Child not in Processed:

Queue.append(Child)

print('Done')

**Vaccum Cleaner**

States = list()

def vcLeft(A,m):

global States

try:

A[m][1] = 0

A[m-1][1] = 1

if A not in States:

States.append(A)

vcOperate(A,m-1)

except:

pass

def vcRight(A,m):

global States

try:

A[m][1] = 0

A[m+1][1] = 1

if A not in States:

States.append(A)

vcOperate(A,m+1)

except:

pass

def vcClean(A,m):

global States

try:

A[m][0] = 0

if A not in States:

States.append(A)

vcOperate(A,m)

except:

pass

def vcOperate(A,m):

vcLeft(A,m)

vcClean(A,m)

vcRight(A,m)

n = int(input("Enter no.of rooms : "))

A = [[1,0] for i in range(n)]

for i in range(n):

vcOperate(A,i)

**Tic-Tac-Toe**

import random

SYSTEM, PLAYER = 'X', 'O'

def isMovesLeft(Board):

for i in range(3):

for j in range(3):

if (Board[i][j] == '\_'):

return True

return False

def evaluate(Board):

for row in range(3) :

if (Board[row][0] == Board[row][1] and Board[row][1] == Board[row][2]):

if (Board[row][0] == SYSTEM):

return 10

elif (Board[row][0] == PLAYER):

return -10

for col in range(3):

if (Board[0][col] == Board[1][col] and Board[1][col] == Board[2][col]):

if (Board[0][col] == SYSTEM):

return 10

elif (Board[0][col] == PLAYER):

return -10

if (Board[0][0] == Board[1][1] and Board[1][1] == Board[2][2]):

if (Board[0][0] == SYSTEM):

return 10

elif (Board[0][0] == PLAYER):

return -10

if (Board[0][2] == Board[1][1] and Board[1][1] == Board[2][0]):

if (Board[0][2] == SYSTEM):

return 10

elif (Board[0][2] == PLAYER):

return -10

return 0

def minimax(Board, depth, isSystem):

score = evaluate(Board)

if (score == 10):

return score

if (score == -10):

return score

if (isMovesLeft(Board) == False):

return 0

if (isSystem):

best = -1000

for i in range(3):

for j in range(3):

if (Board[i][j]=='\_'):

Board[i][j] = SYSTEM

best = max(best, minimax(Board, depth + 1, not isSystem))

Board[i][j] = '\_'

return best

else :

best = 1000

for i in range(3):

for j in range(3):

if (Board[i][j] == '\_'):

Board[i][j] = PLAYER

best = min(best, minimax(Board, depth + 1, not isSystem))

Board[i][j] = '\_'

return best

def findBestMove(Board):

bestVal = -1000

bestMove = (-1, -1)

for i in range(3):

for j in range(3):

if (Board[i][j] == '\_'):

Board[i][j] = SYSTEM

moveVal = minimax(Board, 0, False)

Board[i][j] = '\_'

if (moveVal > bestVal):

bestMove = (i, j)

bestVal = moveVal

return bestMove

Board = [['\_', '\_', '\_'], ['\_', '\_', '\_'], ['\_', '\_', '\_']]

isSystem = random.choice([False,True])

while True:

if isSystem:

(x,y) = findBestMove(Board)

print("Systems's move : ",x,y)

Board[x][y] = SYSTEM

else:

x,y = map(int, input("Player's move : ").split())

if Board[x][y]!='\_':

print("Enter a valid move.")

isSystem = not isSystem

else:

Board[x][y] = PLAYER

for row in Board:

print(row)

score = evaluate(Board)

if score==10:

print("It's System win!")

break

elif score==-10:

print("It's Player win!")

break

elif not isMovesLeft(Board):

print("It's a Draw!")

break

else:

isSystem = not isSystem

**Hill Climbing**

v = int(input("Enter no.of nodes : "))

e = int(input("Enter no.of edges : "))

graph = [[0 for i in range(v)] for j in range(v)]

for i in range(e):

a,b,w = map(int, input("Enter edge points and weight : ").split())

graph[a][b] = w

graph[b][a] = w

# graph = [[0,20,40,22],[20,0,13,30],[40,13,0,12],[22,30,12,0]]

print(graph)

start = int(input("Start node : "))

key = start

Processed = []

select = -1

while True:

if len(Processed)==v or select==start:

break

available = []

mx = 1e9

select = -1

for i in range(v):

if graph[key][i]!=0 and mx>graph[key][i] and i not in Processed:

select = i

mx = graph[key][i]

Processed.append(select)

key = select

print(\*([start]+Processed), sep=" --> ")