1.Program to implement tic-tac-toe game playing

source code:

import os

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

player=1

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(" | | ")

def checkposition(x):

if board[x]==" " :

return True

else:

return False

def checkwin():

global game

if (board[1]==board[2] and board[3]==board[2] 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[9]!=" "):

game=win

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

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

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("player 1 [x] ---- player 2 [0]\n")

print()

print()

print("please wait")

t=0

while game==running:

t+=1

os.system('cls')

drawboard()

if player%2==0:

print("player2's chance")

mark='x'

else:

print("player1's chance")

mark='0'

choice=int(input("enter the position b/w [1-9] where u want to mark:"))

if checkposition(choice):

board[choice]=mark

player+=1

checkwin()

drawboard()

if game==draw:

print("game draw")

elif game==win:

player-=1

if player%2!=0:

print("player1 won")

else:

print("player2 won")

2.Program to implement water-jug problem

source code:

x=0

y=0

m=5

n=3

print("initial state = (0,0)")

print("capacities=(5,3)")

print("goal state =(x,y)")

while x!=4:

r=int(input("enter rule number"))

if r==1:

x=m

print(x,y)

elif r==2:

x=0

print(x,y)

elif r==3:

y=n

print(x,y)

elif r==4:

y=0

print(x,y)

elif r==5:

x+=y

y=0

print(x,y)

elif r==6:

y+=x

x=0

print(x,y)

elif r==7:

t=m-x

x=m

y-=t

print(x,y)

elif r==8:

t=n-y

y=n

x-=t

print(x,y)

print('(',x,',',y,')')

if x==y:

print("goal state reached : ")

3.Program to implement Breadth-First Search.

source code:

from collections import deque

class Graph:

def \_\_init\_\_(self,directed=True):

self.edges={}

self.directed=directed

def add\_edges(self,node1,node2,\_\_reversed=False):

try:

neighbours=self.edges[node1]

except KeyError:

neighbours=set()

neighbours.add(node2)

self.edges[node1]=neighbours

if not self.directed and not \_\_reversed:

self.add\_edges(node2,node1,True)

def neighbours(self,node):

try:

return self.edges[node]

except KeyError:

return []

def breadth\_first\_search(self,start,goal):

found,fringe,visited,came\_from=False,deque([start]),set([start]),{start:None}

print('{:11s} | {}'.format('Expand Node','fringe'))

print("--------------")

print('{:11s}| {}'.format('-',start))

while not found and len(fringe):

current=fringe.pop()

print('{:11s}'.format(current),end='|')

if current==goal:

found=True

break

for node in self.neighbours(current):

if node not in visited:

visited.add(node)

fringe.appendleft(node)

came\_from[node]=current

print(','.join(fringe))

if found:

print()

return came\_from

else:

print('No path from {} to {}'.format(start,goal))

@staticmethod

def print\_path(came\_from,goal):

parent=came\_from[goal]

if parent:

Graph.print\_path(came\_from,parent)

else:

print(goal,end="")

return

print('=>',goal,end="")

def \_\_str\_\_(self):

return str(self.edges)

graph=Graph(directed=False)

graph.add\_edges('A','B')

graph.add\_edges('A','S')

graph.add\_edges('S','G')

graph.add\_edges('S','C')

graph.add\_edges('C','F')

graph.add\_edges('G','F')

graph.add\_edges('C','D')

graph.add\_edges('C','E')

graph.add\_edges('E','H')

graph.add\_edges('G','H')

start,goal='A','H'

traced\_path=graph.breadth\_first\_search(start,goal)

if (traced\_path):

print('path:',end=" ")

graph.print\_path(traced\_path,goal);

print()

4.Program to implement Depth-First Search

source code:

from collections import deque

class Graph:

def \_\_init\_\_(self,directed=True):

self.edges={}

self.directed=directed

def add\_edges(self,node1,node2,\_\_reversed=False):

try:

neighbours=self.edges[node1]

except KeyError:

neighbours=set()

neighbours.add(node2)

self.edges[node1]=neighbours

if not self.directed and not \_\_reversed:

self.add\_edges(node2,node1,True)

def neighbours(self,node):

try:

return self.edges[node]

except KeyError:

return []

def depth\_first\_search(self,start,goal):

found,fringe,visited,came\_from=False,deque([start]),set([start]),{start:None}

print('{:11s} | {}'.format('Expand Node','fringe'))

print("--------------")

print('{:11s}| {}'.format('-',start))

while not found and len(fringe):

current=fringe.pop()

print('{:11s}'.format(current),end='|')

if current==goal:

found=True

break

for node in self.neighbours(current):

if node not in visited:

visited.add(node)

fringe.append(node)

came\_from[node]=current

print(','.join(fringe))

if found:

print()

return came\_from

else:

print('No path from {} to {}'.format(start,goal))

@staticmethod

def print\_path(came\_from,goal):

parent=came\_from[goal]

if parent:

Graph.print\_path(came\_from,parent)

else:

print(goal,end="")

return

print('=>',goal,end="")

def \_\_str\_\_(self):

return str(self.edges)

graph=Graph(directed=False)

graph.add\_edges('A','B')

graph.add\_edges('A','S')

graph.add\_edges('S','G')

graph.add\_edges('S','C')

graph.add\_edges('C','F')

graph.add\_edges('G','F')

graph.add\_edges('C','D')

graph.add\_edges('C','E')

graph.add\_edges('E','H')

graph.add\_edges('G','H')

start,goal='A','H'

traced\_path=graph.depth\_first\_search(start,goal)

if (traced\_path):

print('path:',end=" ")

graph.print\_path(traced\_path,goal);

print()

5.Program to implement Branch and Bound Search

source code:

#Branch And Bound NQueens Problem

N=int(input("enter board size\n"))

def printSol(board):

for i in range(N):

for j in range(N):

print(board[i][j],end=" ")

print()

def isSafe(row,col,slashcode,backslashcode,rowlookup,slashcodelookup,backslashcodelookup):

if slashcodelookup[slashcode[row][col]] or backslashcodelookup[backslashcode[row][col]] or rowlookup[row]:

return False

return True

def solveNQueensUtil(board,col,slashcode,backslashcode,rowlookup,slashcodelookup,backslashcodelookup):

if col>=N:

return True

for i in range(N):

if isSafe(i,col,slashcode,backslashcode,rowlookup,slashcodelookup,backslashcodelookup):

board[i][col]=1

rowlookup[i]=True

slashcodelookup[slashcode[i][col]]=True

backslashcodelookup[backslashcode[i][col]]=True

if solveNQueensUtil(board,col+1,slashcode,backslashcode,rowlookup,slashcodelookup,backslashcodelookup):

return True

board[i][col]=0

rowlookup[i]=False

slashcodelookup[slashcode[i][col]]=False

backslashcodelookup[backslashcode[i][col]]=False

return False

def solveNQueens():

board=[[0 for i in range(N)]for j in range(N)]

slashcode=[[0 for i in range(N)]for j in range(N)]

backslashcode=[[0 for i in range(N)]for j in range(N)]

rowlookup=[False]\*N

x=2\*N-1

slashcodelookup=[False]\*x

backslashcodelookup=[False]\*x

for rr in range(N):

for cc in range(N):

slashcode[rr][cc]=rr+cc

backslashcode[rr][cc]=rr-cc+N-1

if solveNQueensUtil(board,0,slashcode,backslashcode,rowlookup,slashcodelookup,backslashcodelookup)==False:

print("solution does not exist")

return False

printSol(board)

return True

solveNQueens()

6.Program to implement A\* Search.

**source code:**

from queue import heappop, heappush

from math import inf

class Graph:

def \_\_init\_\_(self, directed=True):

self.edges = {}

self.huristics = {}

self.directed = directed

def add\_edge(self, node1, node2, cost = 1, \_\_reversed=False):

try: neighbors = self.edges[node1]

except KeyError: neighbors = {}

neighbors[node2] = cost

self.edges[node1] = neighbors

if not self.directed and not \_\_reversed: self.add\_edge(node2, node1, cost, True)

def set\_huristics(self, huristics={}):

self.huristics = huristics

def neighbors(self, node):

try: return self.edges[node]

except KeyError: return []

def cost(self, node1, node2):

try: return self.edges[node1][node2]

except: return inf

def a\_star\_search(self, start, goal):

found, fringe, visited, came\_from, cost\_so\_far = False, [(self.huristics[start], start)], set([start]), {start: None}, {start: 0}

print('{:11s} | {}'.format('Expand Node', 'Fringe'))

print('--------------------')

print('{:11s} | {}'.format('-', str(fringe[0])))

while not found and len(fringe):

\_, current = heappop(fringe)

print('{:11s}'.format(current), end=' | ')

if current == goal: found = True; break

for node in self.neighbors(current):

new\_cost = cost\_so\_far[current] + self.cost(current, node)

if node not in visited or cost\_so\_far[node] > new\_cost:

visited.add(node); came\_from[node] = current; cost\_so\_far[node] = new\_cost

heappush(fringe, (new\_cost + self.huristics[node], node))

print(', '.join([str(n) for n in fringe]))

if found: print(); return came\_from, cost\_so\_far[goal]

else: print('No path from {} to {}'.format(start, goal)); return None, inf

@staticmethod

def print\_path(came\_from, goal):

parent = came\_from[goal]

if parent:

Graph.print\_path(came\_from, parent)

else: print(goal, end='');return

print(' =>', goal, end='')

def \_\_str\_\_(self):

return str(self.edges)

graph = Graph(directed=True)

graph.add\_edge('A', 'B', 4)

graph.add\_edge('A', 'C', 1)

graph.add\_edge('B', 'D', 3)

graph.add\_edge('B', 'E', 8)

graph.add\_edge('C', 'C', 0)

graph.add\_edge('C', 'D', 7)

graph.add\_edge('C', 'F', 6)

graph.add\_edge('D', 'C', 2)

graph.add\_edge('D', 'E', 4)

graph.add\_edge('E', 'G', 2)

graph.add\_edge('F', 'G', 8)

graph.set\_huristics({'A': 8, 'B': 8, 'C': 6, 'D': 5, 'E': 1, 'F': 4, 'G': 0})

start, goal = 'A', 'G'

traced\_path, cost = graph.a\_star\_search(start, goal)

if (traced\_path): print('Path:', end=' '); Graph.print\_path(traced\_path, goal); print('\nCost:', cost)