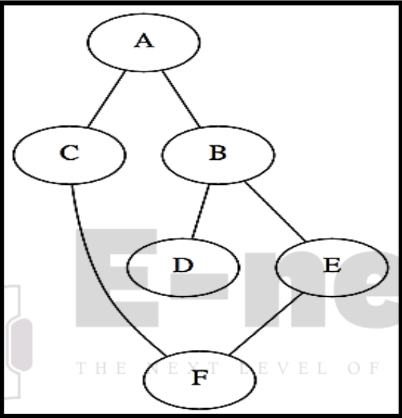
Practical No: 1A

Aim: Write a program to implement depth first search algorithm.

Diagram:



Code:

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(graph, node, visited):

if node not in visited:

visited.append(node)

for n in graph[node]:

dfs(graph,n, visited)

return visited

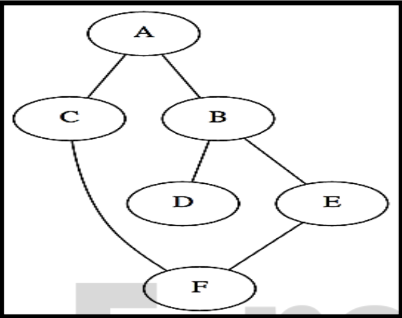
visited = dfs(graph1,'A', [])

print(visited)

Practical No:1B

Aim: Write a program to implement breadth first search algorithm.

Diagram:



Code:

graph = {'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 bfs(start):

queue = [start]

levels={}

levels[start]=0

visited = set(start)

while queue:

node = queue.pop(0)

neighbours=graph[node]

for neighbor in neighbours:

if neighbor not in visited:

queue.append(neighbor)

visited.add(neighbor)

levels[neighbor]= levels[node]+1

print(levels)

return visited

print(str(bfs('A')))

def bfs\_paths(graph, start, goal):

queue = [(start, [start])]

while queue:

(vertex, path) = queue.pop(0)

for next in graph[vertex] - set(path):

if next == goal:

yield path + [next]

else:

queue.append((next, path + [next]))

result=list(bfs\_paths(graph, 'A', 'F'))

print(result)

def shortest\_path(graph, start, goal):

try:

return next(bfs\_paths(graph, start, goal))

except StopIteration:

return None

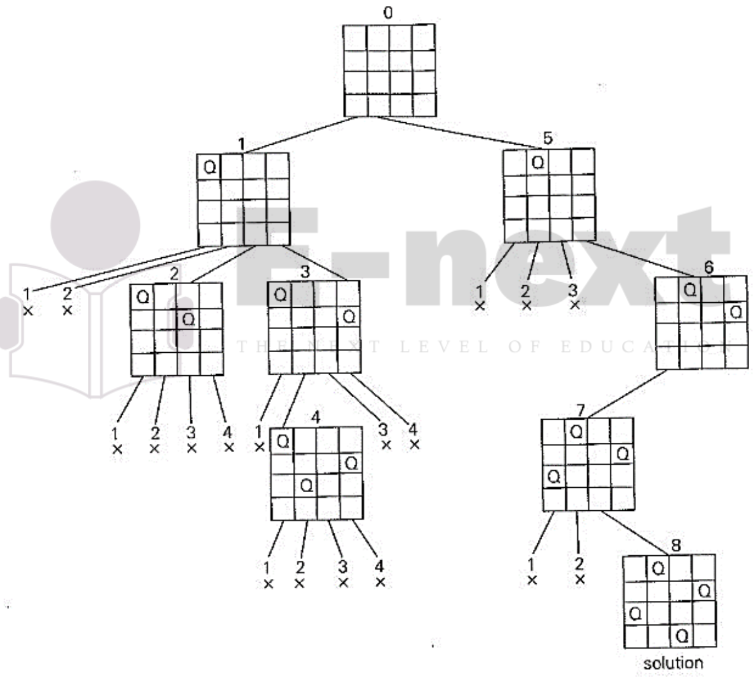
result1=shortest\_path(graph, 'A', 'F')

print(result1)

Practical No: 2A

Aim: Write a program to simulate 4-Queen / N-Queen problem.

Diagram:



Code:

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)

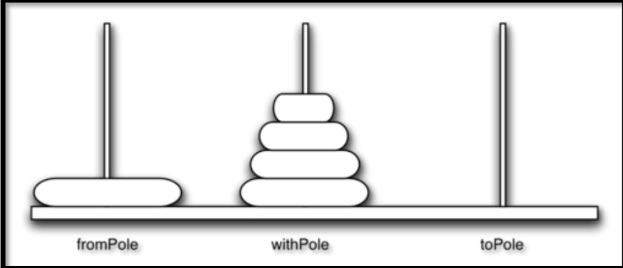
n = int(input('Enter n: '))

solve\_queen(n)

Practical No: 2B

Aim: Write a program to solve tower of Hanoi problem.

Diagram:

Code:

Code:

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("moving disk from",fp,"to",tp)

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

Practical No: 3A

Aim: Write a program to implement A\* algorithm.

Note: Install 2 package in python scripts directory using pip command.

1.pip install simpleai

2.pip install pydot flask

Code:

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())

Practical No: 3B

Aim: Write a program to implements Ao\* algorithm.

Code:

import queue as Q

from RMP import dict\_gn

from RMP import dict\_hn

start='Arad'

goal='Bucharest'

result = ''

def get\_fn(citystr):

cities=citystr.split(" , ")

hn=gn=0

for ctr in range(0, len(cities)-1):

gn=gn+dict\_gn[cities[ctr]][cities[ctr+1]]

hn=dict\_hn[cities[len(cities)-1]]

return(hn+gn)

def expand(cityq):

global result

tot, citystr, thiscity=cityq.get()

if thiscity==goal:

result=citystr+" : : "+str(tot)

return

for cty in dict\_gn[thiscity]:

cityq.put((get\_fn(citystr+" , "+cty), citystr+" , "+cty, cty))

expand(cityq)

def main():

cityq=Q.PriorityQueue()

thiscity=start

cityq.put((get\_fn(start),start,thiscity))

expand(cityq)

print("The A\* path with the total is: ")

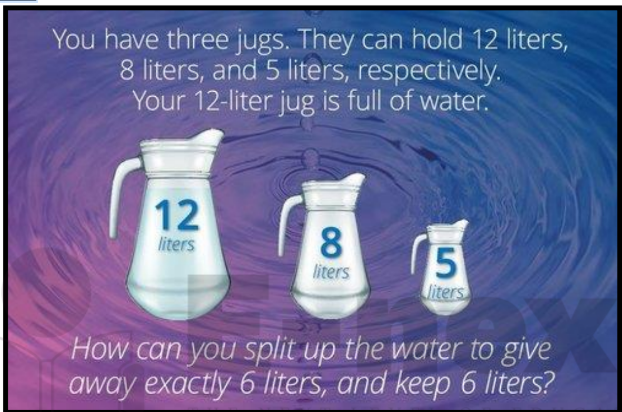
print(result)

main()

Practical No: 4A

Aim: Write a program to solve water jug problem.

Diagram:



Code:

capacity = (12,8,5)

x = capacity[0]

y = capacity[1]

z = capacity[2]

memory = {}

ans = []

def get\_all\_states(state):

a = state[0]

b = state[1]

c = state[2]

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

ans.append(state)

return True

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

return False

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

if(a>0):

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

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

if(b>0):

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

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):

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

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()

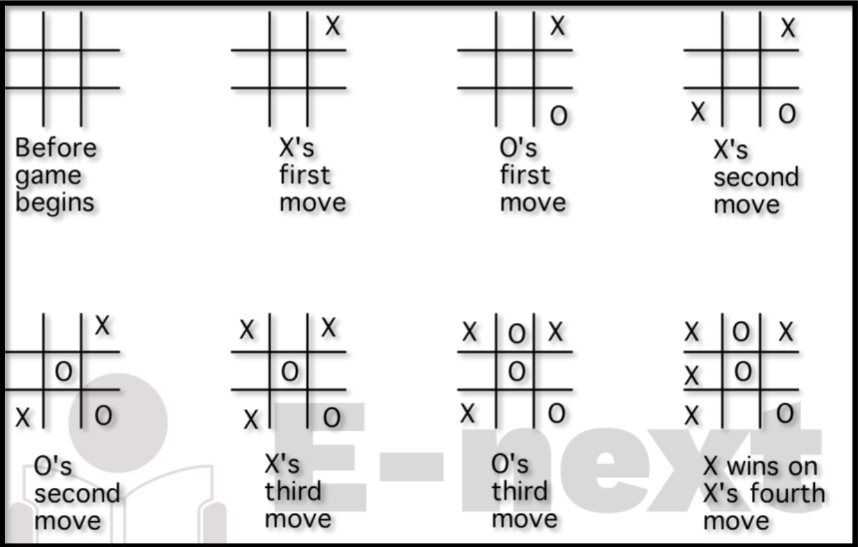
for i in ans:

print(i)

Practical No:4B

Aim: Design the simulation of TIC – TAC –TOE game using min-max algorithm.

Design:



Code:

import os

import time

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[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

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()

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")

Practical No:5

Aim : Design an application to simulate number puzzle problem.

Code:

from \_\_future\_\_ import print\_function

from simpleai.search import astar, SearchProblem

from simpleai.search.viewers import WebViewer

GOAL = '''1-2-3

4-5-6

7-8-e'''

INITIAL = '''4-1-2

7-e-3

8-5-6'''

def list\_to\_string(list\_):

return '\n'.join(['-'.join(row) for row in list\_])

def string\_to\_list(string\_):

return [row.split('-') for row in string\_.split('\n')]

def find\_location(rows, element\_to\_find):

'''Find the location of a piece in the puzzle.

Returns a tuple: row, column'''

for ir, row in enumerate(rows):

for ic, element in enumerate(row):

if element == element\_to\_find:

return ir, ic

# we create a cache for the goal position of each piece, so we don't have to

# recalculate them every time

goal\_positions = {}

rows\_goal = string\_to\_list(GOAL)

for number in '12345678e':

goal\_positions[number] = find\_location(rows\_goal, number)

class EigthPuzzleProblem(SearchProblem):

def actions(self, state):

'''Returns a list of the pieces we can move to the empty space.'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

actions = []

if row\_e > 0:

actions.append(rows[row\_e - 1][col\_e])

if row\_e < 2:

actions.append(rows[row\_e + 1][col\_e])

if col\_e > 0:

actions.append(rows[row\_e][col\_e - 1])

if col\_e < 2:

actions.append(rows[row\_e][col\_e + 1])

return actions

def result(self, state, action):

'''Return the resulting state after moving a piece to the empty space.

(the "action" parameter contains the piece to move)

'''

rows = string\_to\_list(state)

row\_e, col\_e = find\_location(rows, 'e')

row\_n, col\_n = find\_location(rows, action)

rows[row\_e][col\_e], rows[row\_n][col\_n] = rows[row\_n][col\_n], rows[row\_e][col\_e]

return list\_to\_string(rows)

def is\_goal(self, state):

'''Returns true if a state is the goal state.'''

return state == GOAL

def cost(self, state1, action, state2):

'''Returns the cost of performing an action. No useful on this problem, i

but needed.

'''

return 1

def heuristic(self, state):

'''Returns an \*estimation\* of the distance from a state to the goal.

We are using the manhattan distance.

'''

rows = string\_to\_list(state)

distance = 0

for number in '12345678e':

row\_n, col\_n = find\_location(rows, number)

row\_n\_goal, col\_n\_goal = goal\_positions[number]

distance += abs(row\_n - row\_n\_goal) + abs(col\_n - col\_n\_goal)

return distance

result = astar(EigthPuzzleProblem(INITIAL))

for action, state in result.path():

print('Move number', action)

print(state)

Practical No: 6

Aim: Write a program to shuffle Deck of cards.

Diagram :



Code:

import random

cardfaces = []

suits = ["Hearts", "Diamonds", "Clubs", "Spades"]

royals = ["J", "Q", "K", "A"]

deck = []

for i in range(2,11):

cardfaces.append(str(i))

for j in range(4):

cardfaces.append(royals[j])

for k in range(4):

for l in range(13):

card = (cardfaces[l] + " of " + suits[k])

deck.append(card)

random.shuffle(deck)

for m in range(52):

print(deck[m])

Practical No:7A

Aim: Write a program to implement alpha beta search

Code:

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)

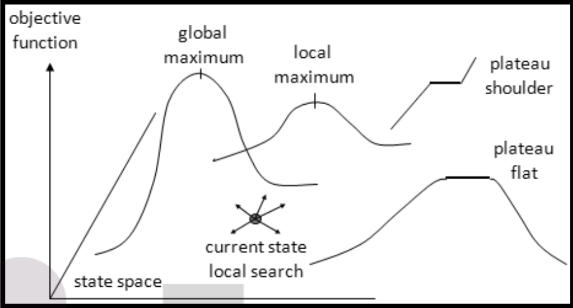
if \_\_name\_\_ == "\_\_main\_\_":

alphabeta(None)

Practical No: 7B

Aim: Write a program for Hill climbing problem.

Diagram:



Code:

import math

increment = 0.1

startingPoint = [1, 1]

point1 = [1,5]

point2 = [6,4]

point3 = [5,2]

point4 = [2,1]

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

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

i+=1

else:

flag = False

Practical No: 8

Aim : Implementation Of Constraints Satisfactions Problem.

Code :

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))