**Q.) BREADTH-FIRST SEARCH**

#create a dictionary(KEY:VALUE pairs) named as graph to store the nodes in your tree

graph = {

'S': ['A', 'B'], #list[] to store the nodes

'A':['C','D'],

'B':['G','H'],

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

'D':[],

'E':['K'],

'F':[],

'G':['I'],

'H':[],

'I':[],

'K':[],

}

visited=[] #List of visited nodes.

queue=[] #initialization of queue

def bfs(visited,graph,node):

visited.append(node)

queue.append(node)

while queue:

m=queue.pop(0)

print(m , end=" ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

queue.append(neighbour)

print("The Result of Breadth First Search is as follows: ")

bfs(visited,graph,'S')

**Q.) DEPTH-FIRST SEARCH**

graph = {

'3':['5','8','25'],

'5':['1','2'],

'8':[],

'25':['12','8'],

'1':[],

'2':[],

'12':['6'],

'6' :['4','9'],

'4':[],

'9':[]

}

visited = set()

def dfs(visited, graph, node):

if node not in visited:

print(node, end=" ")

visited.add(node)

for neighbour in graph[node]:

dfs(visited,graph,neighbour)

dfs(visited,graph,'3')

**Q.) N-QUEEN / 4-QUEEN**

N = int(input("Enter the number of queens"))

board = [[0] \* N for \_ in range(0, N)]

def isattack(i, j):

for k in range(0, N):

if board[i][k] == 1 or board[k][j] == 1:

return True

for k in range(0, N):

for l in range(0, N):

if k + l == i + j or k - l == i - j:

if board[k][l] == 1:

return True

return False

def N\_queen(n):

if n == 0:

return True

for i in range(0, N):

for j in range(0, N):

if not (isattack(i, j)) and board[i][j] != 1:

board[i][j] = 1

if N\_queen(n - 1):

return True

board[i][j] = 0

return False

if N\_queen(N):

for i in board:

print(i)

else:

print("Solution not found")

**Q.) TOWER OF HANOI**

def TowerofHanoi(n,s\_pole,d\_pole,i\_pole):

if n==1:

print("Move Disc 1 From Pole",s\_pole,"to pole",d\_pole)

return

TowerofHanoi(n-1,s\_pole,i\_pole,d\_pole)

print("Move Disc",n,"from pole",s\_pole,"to pole",d\_pole)

TowerofHanoi(n-1,i\_pole,d\_pole,s\_pole)

n=3

TowerofHanoi(n,'A','C','B')

**Q.) ALPHA BETA PRUINING**

MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):

if depth == 3:

return values[nodeIndex]

if maximizingPlayer:

best = MIN

# Recur for left & right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, False, values, alpha, beta)

best = max(best, val)

alpha = max(alpha, best)

# Alpha-Beta Pruning

if beta <= alpha:

break

return best

else:

best = MAX

# Recur for left and right children

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex \* 2 + i, True, values, alpha, beta)

best = min(best, val)

beta = min(beta, best)

# Alpha-Beta Pruning

if beta <= alpha:

break

return best

# Driver Code

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

values = [3, 5, 6, 9, 1, 2, 0, -1]

print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

**Q.) WATERJUG**

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

#empty jug c

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)

**Q.) MISSIONARIES AND CANNIBALS**

from collections import deque

initial\_state = (3, 3, 1)

goal\_state = (0, 0, 0)

def is\_valid(state):

missionaries\_left, cannibals\_left, boat\_location = state

if (

missionaries\_left < 0

or cannibals\_left < 0

or missionaries\_left > 3

or cannibals\_left > 3

):

return False

if (

(missionaries\_left < cannibals\_left and missionaries\_left > 0)

or (3 - missionaries\_left < 3 - cannibals\_left and 3 - missionaries\_left > 0)

):

return False

return True

def get\_next\_states(state):

states = []

missionaries\_left, cannibals\_left, boat\_location = state

passengers = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

for m, c in passengers:

new\_missionaries\_left = missionaries\_left - m \* boat\_location

new\_cannibals\_left = cannibals\_left - c \* boat\_location

new\_boat\_location = 1 - boat\_location

new\_state = (new\_missionaries\_left, new\_cannibals\_left, new\_boat\_location)

if is\_valid(new\_state):

states.append(new\_state)

return states

def solve():

visited = set()

queue = deque([(initial\_state, [])])

while queue:

current\_state, path = queue.popleft()

if current\_state == goal\_state:

return path + [current\_state]

visited.add(current\_state)

next\_states = get\_next\_states(current\_state)

for next\_state in next\_states:

if next\_state not in visited:

queue.append((next\_state, path + [current\_state]))

return None

solution = solve()

if solution:

print("Solution:")

for i, state in enumerate(solution):

print(f"Step {i + 1}: {state}")

else:

print("No solution found")

**Q.) SHUFFLE DECK OF CARDS**

import random

ranks =['2','3','4','5','6','7','8','9','10','jack','queen','king','ace']

suits=['hearts','diamonds','clubs','spades']

deck=[{'rank':rank,'suit':suit} for rank in ranks for suit in suits]

def shuffle\_deck(deck):

random.shuffle(deck)

def display\_deck(deck):

for card in deck:

print(f"{card['rank']} of {card['suit']}")

shuffle\_deck(deck)

print("shuffles deck:")

display\_deck(deck)

**Q.) ASSOCIATIVE**

associative\_addition(A,B,C,Result):- Result is A+(B+C).

associative\_multiplication(A,B,C,Result):-Result is A\*(B\*C).

**Q.) DISTRIBUTIVE**

distributive\_multiplication\_over\_add(A,B,C,Result):-Result is A\*(B+C).

distributive\_add\_over\_multiplication(A,B,C,Result):-Result is A+(B\*C).

**Q.) PREDICATES EXAMPLE**

**female (pam).**

**female (liz).**

**female (pat).**

**female (ann).**

**male (tom).**

**male (bob).**

**male (tom).**

**male (bob).**

**male (peter).**

**parent (pam, bob).**

**parent (tom, bob).**

**parent (tom, liz).**

**parent (bob, ann).**

**parent (bob, pat).**

**parent (pat, jim)**

**parent (tom, peter).**

**Defining Rules**

**mother (X, Y) :- parent (X, Y), female (X).**

**father (X, Y) :- parent (X, Y), male (X).**

**sister (X, Y) :- parent (Z,X), parent (Z,Y), female (X), X \== Y.**

**brother (X, Y): parent (Z,X), parent (Z,Y), male (X), X\==Y.**

**grandmother (X,Z) :- mother (X, Y), parent (Y, Z).**

**grandfather (X,Z) :- father (X, Y), parent (Y, Z).**

**uncle (X, Z) :- brother (X, Y), parent (Y, Z).**

**aunt (X, Z) :- sister (X, Y), parent (Y,Z).**

**Q.) BLOCK WORLD**

class BlockWorld:

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

self.state = ['A', 'B', 'C'] #Initial state: Three blocks on table

def move(self, source, destination):

if source == destination or source not in self.state:

return False #Invalid move

block = self.state.pop(self.state.index(source)) #Remove block from source

self.state.append(block) #place the block on destination

return True

def solve(self):

#Move A to C

self.move('A','C')

#Move A to B

self.move('A','B')

#Move C to B

self.move('C','B')

#Move A to C

self.move('A','C')

#Move B to A

self.move('B','A')

#Move B to C

self.move('B','C')

#Move A to C

self.move('A','C')

return self.state

# Create a BlockWorld instance

block\_world = BlockWorld()

#Solve the problem

final\_state = block\_world.solve()

print("Final state:", final\_state)

**Q.) A\***

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):

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())