

**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**  
“JnanaSangama”, Belgaum -590014, Karnataka.



**LAB REPORT**  
**On**

**ARTIFICIAL INTELLIGENCE**

**Submitted by**

**JAGADEESH A LATTI (1BM21CS079)**

**in partial fulfillment for the award of the degree of  
BACHELOR OF ENGINEERING  
in  
COMPUTER SCIENCE AND ENGINEERING**



**B.M.S. COLLEGE OF ENGINEERING  
(Autonomous Institution under VTU)  
BENGALURU-560019  
Oct 2023-Feb 2024**

**B. M. S. College of Engineering,  
Bull Temple Road, Bangalore 560019  
(Affiliated To Visvesvaraya Technological University, Belgaum)  
Department of Computer Science and Engineering**



**CERTIFICATE**

This is to certify that the Lab work entitled "**ARTIFICIAL INTELLIGENCE**" carried out by **JAGADEESH A LATTI (1BM21CS079)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence Lab - **(22CS5PCAIN)** work prescribed for the said degree.

**Swathi Sridharan**  
Assistant Professor  
Department of CSE  
BMSCE, Bengaluru

**Dr. Jyothi S Nayak**  
Professor and Head  
Department of CSE  
BMSCE, Bengaluru

## Table of Contents

<b>SL No</b>	<b>Name of Experiment</b>	<b>Page No</b>
1	Implement Tic –Tac –Toe Game	1-7
2	Implement 8 puzzle problem	8-12
3	Implement Iterative deepening search algorithm.	13-17
4	Implement A* search algorithm.	17-23
5	Implement vaccum cleaner agent.	23-27
6	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .	28-30
7	Create a knowledge base using prepositional logic and prove the given query using resolution	30-35
8	Implement unification in first order logic	36-41
9	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	42-46
10	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	46-52

## 1. Implement Tic - Tac - Toe Game.

Date \_\_\_\_\_  
Page \_\_\_\_\_

```
if (xstate[choice] != 1 and zstate[choice] != 1):
```

```
    xstate[choice] = 1
```

```
    count += 1
```

```
else:
```

```
    print ("Already this space is occupied,  
try again")
```

```
    continue
```

```
win = checkwin(xstate, zstate)
```

```
if (win == -1):
```

```
    print ("Match over")
```

```
    pointboard(xstate, zstate)
```

```
    over = False
```

```
    break
```

```
if (count == 9):
```

```
    print ("Match over")
```

```
    print ("No space left")
```

```
    over = False
```

```
    break
```

```
    turn = (-turn)
```

Algorithm:

1) sum(a, b, c) :

↳ This function takes 3 inputs & return their sum

2) pointboard(xstate, zstate) :

↳ It prints tic-tac-toe board based on current state of X & Z

↳ It converts the state of each cell to 'X' if X has marked it, 'O' if Z has marked it

it or else ~~the~~ cell number.

### 3) checkwin(xstate, zstate) {

    ↳ It checks who won the game

    ↳ It checks all possible winning combinations

    ↳ If X wins, it prints & returns 1

    ↳ If Z wins, it prints & returns 0

    ↳ If there is no winner, it returns -1

4. The game runs in a loop until it is explicitly terminated.

    ↳ The game alternates turns between X & Z until there is a winner or a draw.

    ↳ Point-based function is called to print current state of board.

    ↳ The checkwin function is called after each move to check if there's winner.

    ↳ Game ends when there is a winner or all spaces are filled.

Output:

0 | 1 | 2

3 | 4 | 5

6 | 7 | 8

X's turn, Enter a number: 0

X | 1 | 2

3 | 4 | 5

6 | 7 | 8

```

tic=[]
import random

def board(tic):
    for i in range(0,9,3):
        print("+"+"-*29+")
        print("|"+ "*9+"|"+ "*9+"|"+ "*9+"|)
        print("|"+ "*3,tic[0+i]," "*3+"|"+ "*3,tic[1+i]," "*3+"|"+ "*3,tic[2+i]," "*3+"|")
        print("|"+ "*9+"|"+ "*9+"|"+ "*9+"|)
    print("+"+"-*29+")

```

```

def update_comp():
    global tic,num
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='X'
            if winner(num-1)==False:
                #reverse the change
                tic[num-1]=num
        else:
            return
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='O'
            if winner(num-1)==True:
                tic[num-1]='X'
            return
        else:

```

```

tic[num-1]=num
num=random.randint(1,9)
while num not in tic:
    num=random.randint(1,9)
else:
    tic[num-1]='X'

def update_user():
    global tic,num
    num=int(input("enter a number on the board :"))
    while num not in tic:
        num=int(input("enter a number on the board :"))
    else:
        tic[num-1]='O'

def winner(num):
    if tic[0]==tic[4] and tic[4]==tic[8] or tic[2]==tic[4] and tic[4]==tic[6]:
        return True
    if tic[num]==tic[num-3] and tic[num-3]==tic[num-6]:
        return True
    if tic[num//3*3]==tic[num//3*3+1] and tic[num//3*3+1]==tic[num//3*3+2]:
        return True
    return False

try:
    for i in range(1,10):
        tic.append(i)
    count=0

```

```
#print(tic)
board(tic)
while count!=9:
    if count%2==0:
        print("computer's turn :")
        update_comp()
        board(tic)
        count+=1
    else:
        print("Your turn :")
        update_user()
        board(tic)
        count+=1
    if count>=5:
        if winner(num-1):
            print("winner is ",tic[num-1])
            break
        else:
            continue
except:
    print("\nerror\n")
```

OUTPUT:

JAGADEESH(1BM21CS079)

[1, 2, 3, 4, 5, 6, 7, 8, 9]

1	2	3
4	5	6
7	8	9

Computer's turn:

1	2	3
4	x	6
7	8	9

Your turn:  
Enter a number on the board: 3

1	2	0
4	x	6
7	8	9

Computer's turn:

1	2	0
x	x	6
7	8	9

YOUR TURN:  
Enter a number on the board: 6

1	2	0
x	x	0
7	8	9

Computer's turn:

1	2	0
x	x	0
7	8	x

Your turn:  
Enter a number on the board: 1

o	2	o
x	x	o
7	8	x

Computer's turn:

o	2	o
x	x	o
7	x	x

Your turn:  
Enter a number on the board: 2

o	o	o
x	x	o

Computer's turn:

o	o	o
x	x	o

X

Winner is X

## 2 .Solve 8 puzzle problems.

3.

8-puzzle:

Initialize the puzzle:

- 1) Create a Puzzle8 class with the initial state, goal state & possible moves

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

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

moves = [(0, 1), (1, 0), (0, -1), (-1, 0)]

- 2) Define methods to print the current state, check if the puzzle is solved, get the index of blank tile & apply move.

Print current state:

for i in range(0, 9, 3):

print(state[i:i+3])

Check if Puzzle is solved:

state == self.goal

Get index of the blank tile (represented by '0')

Apply Move function:

Implement a method to apply a move to the current state, swapping the blank tile with adjacent tile.

## BFS Algorithm

Initialize visited set to keep track of puzzle state to avoid revisiting.

visited = set()

Create an empty queue. It stores tuples where each tuple consists of puzzle state & corresponding path taken to reach state.

queue = Queue()

queue.put((state, initial\_state, [ ]))

until queue is empty over the loop:

i) Deque a state & its path.

ii) Check if the state is the goal state.

if yes print solution path & break out of loop.

iii) if ~~state~~ tile not visited

mark it as visited

iv) get index of blank tile

v) Apply all moves & enqueue the resulting states & paths.

initial state: [1, 2, 3, 7, 5, 6, 0, 4]

queue: [(newstate-1, [move-1]), (newstate-2[move-2])]

for q in

```

def bfs(src,target):
    queue=[]
    queue.append(src)
    exp=[]
    while len(queue)>0:
        source=queue.pop(0)
        #print("queue",queue)
        exp.append(source)

        print(source[0],',',source[1],',',source[2])
        print(source[3],',',source[4],',',source[5])
        print(source[6],',',source[7],',',source[8])
        print("-----")
        if source==target:
            print("Success")
            return

    poss_moves_to_do=[]
    poss_moves_to_do=possible_moves(source,exp)
    #print("possible moves",poss_moves_to_do)
    for move in poss_moves_to_do:
        if move not in exp and move not in queue:
            #print("move",move)
            queue.append(move)

def possible_moves(state,visited_states):
    b=state.index(0)

    #direction array
    d=[]
    if b not in [0,1,2]:

```

```

d.append('u')

if b not in [6,7,8]:
    d.append('d')

if b not in [0,3,6]:
    d.append('l')

if b not in [2,5,8]:
    d.append('r')

pos_moves_it_can=[]

for i in d:
    pos_moves_it_can.append(gen(state,i,b))

return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in
visited_states]

def gen(state,m,b):
    temp=state.copy()
    if m=='d':
        temp[b+3],temp[b]=temp[b],temp[b+3]
    if m=='u':
        temp[b-3],temp[b]=temp[b],temp[b-3]
    if m=='l':
        temp[b-1],temp[b]=temp[b],temp[b-1]
    if m=='r':
        temp[b+1],temp[b]=temp[b],temp[b+1]
    return temp

src=[1,2,3,4,5,6,0,7,8]
target=[1,2,3,4,5,6,7,8,0]
bfs(src,target)

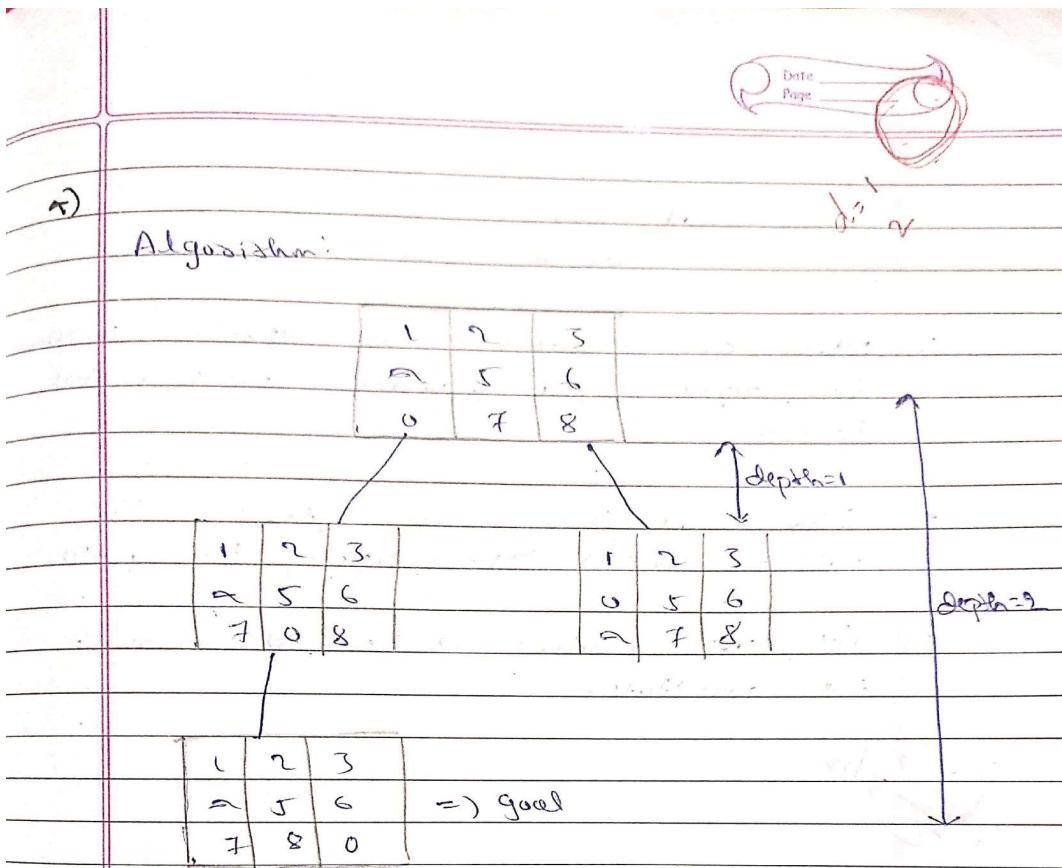
```

## OUTPUT

JAGADEESH(1BM21CS079)

```
└─ 1 | 2 | 3
  4 | 5 | 6
  0 | 7 | 8
-----
 1 | 2 | 3
 0 | 5 | 6
 4 | 7 | 8
-----
 1 | 2 | 3
 4 | 5 | 6
 7 | 0 | 8
-----
 0 | 2 | 3
 1 | 5 | 6
 4 | 7 | 8
-----
 1 | 2 | 3
 5 | 0 | 6
 4 | 7 | 8
-----
 1 | 2 | 3
 4 | 0 | 6
 7 | 5 | 8
-----
 1 | 2 | 3
 4 | 5 | 6
 7 | 8 | 0
-----
Success
```

### 3. Implement Iterative deepening search algorithm.



Algorithm:

- 1) Initialize the initial state = [ ] & goal state for the 8-puzzle.  
goal state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
- 2) set the depth=1 & expand the initial state  
The depth-limited-search(depth) is performed  
if node-state = goal  
return node  
else  
for neighbour in get-neighbour(node.state)  
child = puzzle-node(neighbour, node)  
result = depth-limited-search(depth-1)

if result == True;  
return result

- 3) After one iteration where depth > 1, increment the depth by 1 of perform depth limited search.
- 4) Here get\_neighbors will generate the possible moves by swapping the '0' tile.
- 5) The path travelled is pointed to reach the goal state.

vector<string> v1

```

def id_dfs(puzzle, goal, get_moves):
    import itertools
    #get_moves -> possible_moves
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move], depth - 1)
                if next_route:
                    return next_route

    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route

def possible_moves(state):
    b = state.index(0) # ) indicates White space -> so b has index of it.
    d = [] # direction
    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')

```

```

d.append('r')

pos_moves = []
for i in d:
    pos_moves.append(generate(state, i, b))
return pos_moves

def generate(state, m, b):
    temp = state.copy()

    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    if m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
    if m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    if m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp

# calling ID-DFS
initial = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

route = id_dfs(initial, goal, possible_moves)

if route:

```

```

print("Success!! It is possible to solve 8 Puzzle problem")
print("Path:", route)
else:
    print("Failed to find a solution")

```

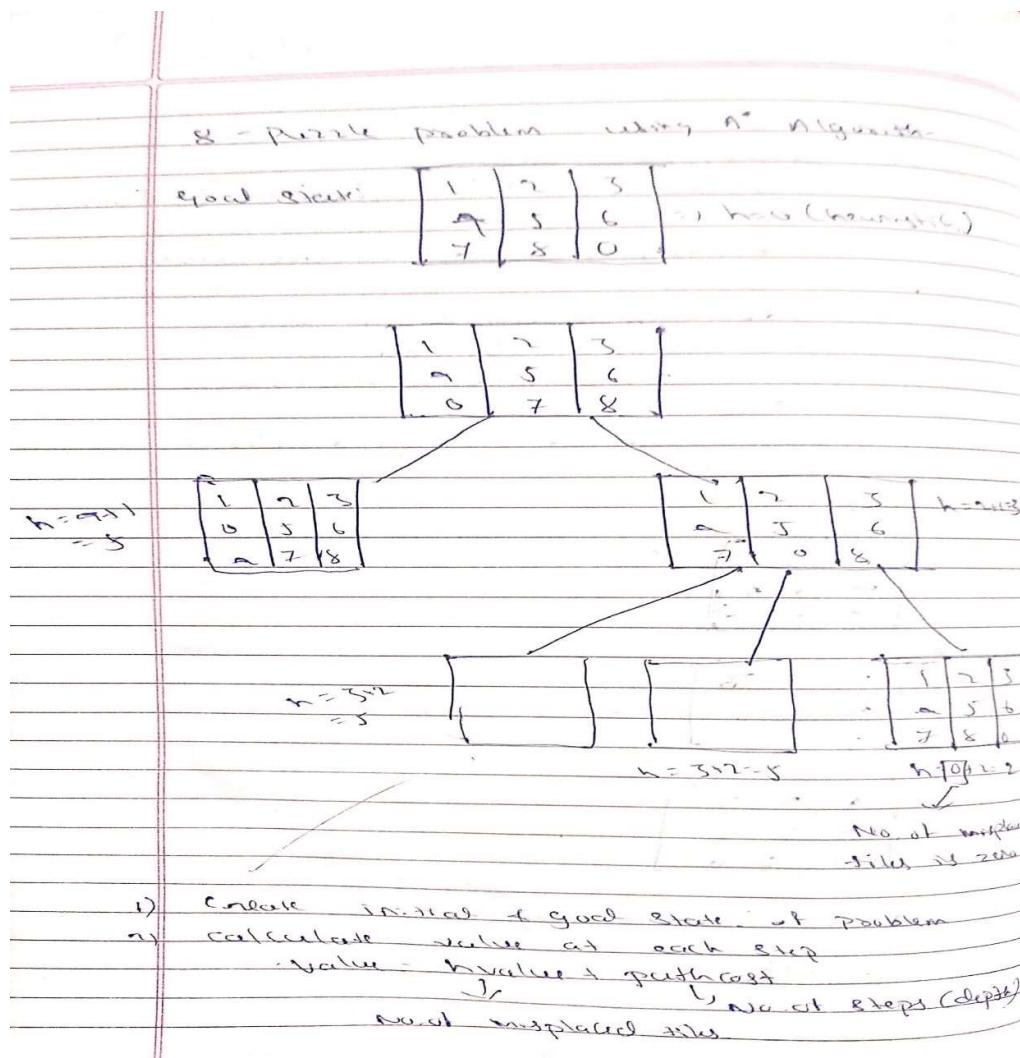
## OUTPUT

```

Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
...Program finished with exit code 0
Press ENTER to exit console.

```

## 4. Implement A\* search algorithm.



3. Go to direction where  $f$ -value is less.

a) All nodes stored in open list (keep track of cell nodes).

b) Explore nodes stored in closed list.

c) Goal is when  $f$ -value is 0, implies that we have reached final state.

(S)  
Goal

```

class Node:

    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

```

```

temp_puz[x1][y1] = temp
return temp_puz

else:
    return None

def copy(self,root):
    """ Copy function to create a similar matrix of the given node"""
    temp = []
    for i in root:
        t = []
        for j in i:
            t.append(j)
        temp.append(t)
    return temp

def find(self,puz,x):
    """ Specifically used to find the position of the blank space """
    for i in range(0,len(self.data)):
        for j in range(0,len(self.data)):
            if puz[i][j] == x:
                return i,j

class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

```

```

def accept(self):
    """ Accepts the puzzle from the user """
    puz = []
    for i in range(0,self.n):
        temp = input().split(" ")
        puz.append(temp)
    return puz

def f(self,start,goal):
    """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
    return self.h(start.data,goal)+start.level

def h(self,start,goal):
    """ Calculates the different between the given puzzles """
    temp = 0
    for i in range(0,self.n):
        for j in range(0,self.n):
            if start[i][j] != goal[i][j] and start[i][j] != '_':
                temp += 1
    return temp

def process(self):
    """ Accept Start and Goal Puzzle state"""
    print("Enter the start state matrix \n")
    start = self.accept()
    print("Enter the goal state matrix \n")
    goal = self.accept()

    start = Node(start,0,0)

```

```

start.fval = self.f(start,goal)

""" Put the start node in the open list"""

self.open.append(start)
print("\n\n")
while True:

    cur = self.open[0]

    print("")
    print(" | ")
    print(" | ")
    print(" \\"/ \n")

    for i in cur.data:

        for j in i:

            print(j,end=" ")
        print("")

    """ If the difference between current and goal node is 0 we have reached the goal
node"""

    if(self.h(cur.data,goal) == 0):

        break

    for i in cur.generate_child():

        i.fval = self.f(i,goal)

        self.open.append(i)

        self.closed.append(cur)

        del self.open[0]

    """ sort the opne list based on f value """
    self.open.sort(key = lambda x:x.fval,reverse=False)

```

puz = Puzzle(3)

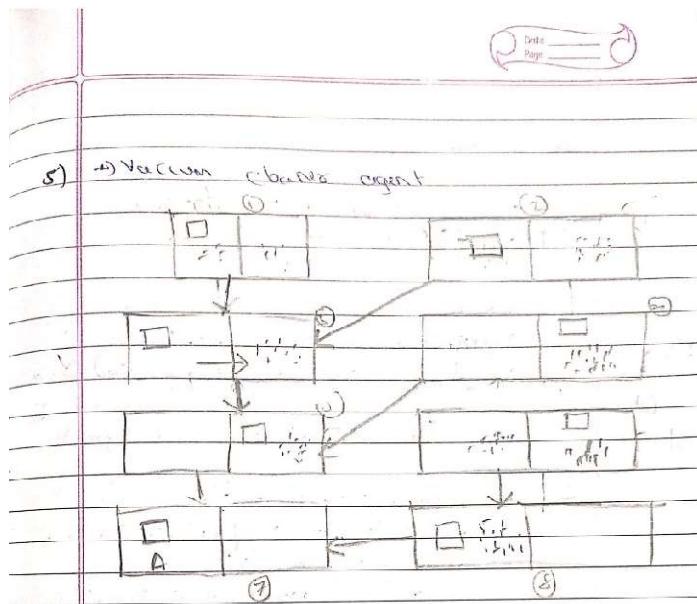
puz.processs

OUTPUT

Success! 8 puzzle problem solved

Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]

## 5. Implement vaccum cleaner agent.



Algorithm

i) Initialize the starting & goal state, the goal is to clean both rooms A & B

ii) If 'status' = Dirty then clean  
else if 'location' = A & 'status' = Clean then return right  
else if 'location' = B & 'status' = Clean then return left

else exit

iii) If both the locations are clean the vaccum cleaner is done with its task.

```
def vacuum_world():
```

```

# 0 indicates Clean and 1 indicates Dirty
goal_state = {'A': '0', 'B': '0'}
cost = 0

location_input = input("Enter Location of Vacuum")
status_input = input("Enter status of " + location_input)
status_input_complement = input("Enter status of other room")

if location_input == 'A':
    # Location A is Dirty.
    print("Vacuum is placed in Location A")
    if status_input == '1':
        print("Location A is Dirty.")
        # suck the dirt and mark it as clean
        cost += 1          #cost for suck
        print("Cost for CLEANING A " + str(cost))
        print("Location A has been Cleaned.")

    if status_input_complement == '1':
        # if B is Dirty
        print("Location B is Dirty.")
        print("Moving right to the Location B. ")
        cost += 1          #cost for moving right
        print("COST for moving RIGHT" + str(cost))
        # suck the dirt and mark it as clean
        cost += 1          #cost for suck
        print("COST for SUCK " + str(cost))
        print("Location B has been Cleaned. ")

    else:
        print("No action" + str(cost))

```

```

# suck and mark clean

print("Location B is already clean.")

if status_input == '0':
    print("Location A is already clean ")

if status_input_complement == '1':# if B is Dirty
    print("Location B is Dirty.")

    print("Moving RIGHT to the Location B. ")
    cost += 1          #cost for moving right
    print("COST for moving RIGHT " + str(cost))

    # suck the dirt and mark it as clean

    cost += 1          #cost for suck
    print("Cost for SUCK" + str(cost))
    print("Location B has been Cleaned. ")

else:
    print("No action " + str(cost))
    print(cost)

    # suck and mark clean

    print("Location B is already clean.")


else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.

    if status_input == '1':
        print("Location B is Dirty.")

        # suck the dirt and mark it as clean
        cost += 1 # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

    if status_input_complement == '1':

```

```

# if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT" + str(cost))

# suck the dirt and mark it as clean

cost += 1 # cost for suck

print("COST for SUCK " + str(cost))

print("Location A has been Cleaned.")

else:

    print(cost)

    # suck and mark clean

    print("Location B is already clean.")

if status_input_complement == '1': # if A is Dirty

    print("Location A is Dirty.")

    print("Moving LEFT to the Location A. ")

    cost += 1 # cost for moving right

    print("COST for moving LEFT " + str(cost))

    # suck the dirt and mark it as clean

    cost += 1 # cost for suck

    print("Cost for SUCK " + str(cost))

    print("Location A has been Cleaned. ")

else:

    print("No action " + str(cost))

    # suck and mark clean

    print("Location A is already clean.")

# done cleaning

```

```
print("GOAL STATE: ")  
print(goal_state)  
print("Performance Measurement: " + str(cost))  
  
print("0 indicates clean and 1 indicates dirty")  
vacuum_world()
```

OUTPUT:

JAGADEESH(1BM21CS079)

```
vacuum>  
→ Enter Location of Vacuum  
Enter status of b1  
Enter status of other room1  
Initial Location Condition{'A': '0', 'B': '0'}  
Vacuum is placed in location B  
Location B is Dirty.  
COST for CLEANING 1  
Location B has been Cleaned.  
Location A is Dirty.  
Moving LEFT to the Location A.  
COST for moving LEFT2  
COST for SUCK 3  
Location A has been Cleaned.  
GOAL STATE:  
{'A': '0', 'B': '0'}  
Performance Measurement: 3
```

**6. Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.**

$P \rightarrow Q$   
 $\neg P \rightarrow Q$

lens  
page

1) Entailment.

INPUT:  
Knowledge Base (KB) in logical form  
Given Statement

Step:

- 1)  $\neg P \rightarrow Q$  is every  
obtain the negation
- 2) Combine with Knowledge base
- 3) Check satisfiability:  
to check if the negation with KB  
is satisfiable the result
- 4) Determine satisfiability.  
if conjunction is not satisfiable  $\rightarrow$  TRUE  
if conjunction is satisfiable  $\rightarrow$  false

$\neg P \rightarrow Q$  is it true in all cases?

from sympy import symbols, And, Not, Implies, satisfiable

```
def create_knowledge_base():
    # Define propositional symbols
    p = symbols('p')
    q = symbols('q')
```

```

r = symbols('r')

# Define knowledge base using logical statements
knowledge_base = And(
    Implies(p, q),      # If p then q
    Implies(q, r),      # If q then r
    Not(r)              # Not r
)

return knowledge_base

def query_entails(knowledge_base, query):
    # Check if the knowledge base entails the query
    entailment = satisfiable(And(knowledge_base, Not(query)))

    # If there is no satisfying assignment, then the query is entailed
    return not entailment

if __name__ == "__main__":
    # Create the knowledge base
    kb = create_knowledge_base()

    # Define a query
    query = symbols('p')

    # Check if the query entails the knowledge base
    result = query_entails(kb, query)

    # Display the results
    print("Knowledge Base:", kb)

```

```
print("Query:", query)
print("Query entails Knowledge Base:", result)
```

OUTPUT:

JAGADEESH(1BM21CS079)

```
→ Enter the knowledge base: (p^q)v(~pvq)
Enter the query: pvq
[True, True, True] :kb= True :q= True
[True, True, False] :kb= True :q= True
[True, False, True] :kb= False :q= True
[True, False, False] :kb= False :q= True
[False, True, True] :kb= True :q= True
[False, True, False] :kb= True :q= True
[False, False, True] :kb= True :q= False
Doesn't entail!!
```

7. Create a knowledge base using propositional logic and prove the given query using resolution

## enquiry entails Knowledge Base - Model

2)

```
def negate_literal(literal)
    if literal[0] = '-':
        return literal[1:]
    else:
        return '+' + literal
```

```
def resolve(C1, C2):
    resolved_clause = set(C1) | set(C2)
```

for literal in C1:

```
    if negate_literal(literal) in C2:
        resolved_clause.remove(literal)
        resolved_clause.remove(negate_literal)
    return tuple(resolved_clause)
```

```
def resolution(knowledge-base):
```

while True:

```
    new_clauses = set()
```

```
    for i, C1 in enumerate(knowledge_base):
```

```
        for j, C2 in enumerate(knowledge_base):
```

```
            if i != j:
```

```
                new_clauses = resolve(C1, C2)
```

if len(new-clause) > 0 & new-clause not  
knowledge base  
new-clause.add(new-clause)

it is not new clause

break

knowledge-base = new-clause  
return knowledge-base

if clause == "original":

kb = {('p', 'g'), ('¬p', 'δ'), ('¬q', '¬r')}

result = resolution(kb)

point("original kb", kb)

point("resolved kb", result)

Output:

Enter statement = Negation

The statement not entailed by knowledge-based

```

import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print("\nStep\t|Clause\t|Derivation\t")
    print('-' * 30)
    i = 1
    for step in steps:
        print(f'{i}.|{step}|{steps[step]}' )
        i += 1

def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ""

def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

split_terms('~PvR')

```

OUTPUT:

JAGADEESH(1BM21CS079)

```
| Enter the clauses separated by a space: p v ~q ~r v p ~q
| Enter the query: ~p
Trying to prove (p)^(v)^(~q)^(~r)^(v)^(p)^(~q)^(~p) by contradiction....
Knowledge Base entails the query, proved by resolution
```

---

```
def contradiction(goal, clause):
    contradictions = [ f'{goal} v {negate(goal)}', f'{negate(goal)} v {goal}' ]
    return clause in contradictions or reverse(clause) in contradictions
```

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
```

```

if negate(c) in terms2:
    t1 = [t for t in terms1 if t != c]
    t2 = [t for t in terms2 if t != negate(c)]
    gen = t1 + t2
    if len(gen) == 2:
        if gen[0] != negate(gen[1]):
            clauses += [f'{gen[0]} v {gen[1]}']
        else:
            if contradiction(goal,f'{gen[0]} v {gen[1]}'):
                temp.append(f'{gen[0]} v {gen[1]}')
            steps[""] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null.\n"
            \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."
            return steps
    elif len(gen) == 1:
        clauses += [f'{gen[0]}']
    else:
        if contradiction(goal,f'{terms1[0]} v {terms2[0]}'):
            temp.append(f'{terms1[0]} v {terms2[0]}')
        steps[""] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null.\n"
        \nA contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."
        return steps
for clause in clauses:
    if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
        temp.append(clause)
        steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'!
        j = (j + 1) % n
        i += 1
return steps

```

```

rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
goal = 'R'
main(rules, goal)

```

Step	Clause	Derivation
1.	Rv~P	Given.
2.	Rv~Q	Given.
3.	~RvP	Given.
4.	~RvQ	Given.
5.	~R	Negated conclusion.
6.		Resolved Rv~P and ~RvP to Rv~R, which is in turn null. A contradiction is found when ~R is assumed as true. Hence, R is true.

```

rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
main(rules, goal)

```

→

Step	Clause	Derivation
1.	PvQ	Given.
2.	~PvR	Given.
3.	~QvR	Given.
4.	~R	Negated conclusion.
5.	QvR	Resolved from PvQ and ~PvR.
6.	PvR	Resolved from PvQ and ~QvR.
7.	~P	Resolved from ~PvR and ~R.
8.	~Q	Resolved from ~QvR and ~R.
9.	Q	Resolved from ~R and QvR.
10.	P	Resolved from ~R and PvR.
11.	R	Resolved from QvR and ~Q.
12.		Resolved R and ~R to Rv~R, which is in turn null. A contradiction is found when ~R is assumed as true. Hence, R is true.

## 8. Implement unification in first order logic :

Date \_\_\_\_\_  
Page \_\_\_\_\_

8. Unification

Step 1: If term 1 is term 2, a unifiable case (constant case)

- if term 1 or term 2 are identical return NIL
- else if term 1 is a variable if term 1 occurs in term 2 return FAIL
- else if term 2 is a variable if term 2 occurs in term 1 return FAIL
- else return  $\alpha(\text{term}_1 // \text{term}_2)$

Step 2: if predicate(term 1) ≠ predicate(term 2)  
return FAIL

Step 3: number of arguments ≠  
return FAIL

Step 4: set (arg 1 to NIL)

Step 5: For i=1 to the number of elements in term 1

- case unify (list term), list term put result into S

$s = \text{FAIL}$

if  $s \neq \text{NIL}$

- a) Apply  $s$  to the remainder of both lists
- b)  $\text{SUBST-APEND}(s, \text{SUBST})$

Step 6: Return  $\text{SUBST}$

Code:

```
import re

def getAttributes(expression):
    import re

def getInitialPredicate(expression):
    return expression.split("(")[0]
```

```

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True

def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]

```

```

def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression

def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isConstant(exp2):
        return [(exp2, exp1)]

    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return False
        else:
            return [(exp2, exp1)]

    if isVariable(exp2):

```

```

if checkOccurs(exp2, exp1):
    return False

else:
    return [(exp1, exp2)]


if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Cannot be unified")
    return False


attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False


head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)
initialSubstitution = unify(head1, head2)
if not initialSubstitution:
    return False

if attributeCount1 == 1:
    return initialSubstitution


tail1 = getRemainingPart(exp1)
tail2 = getRemainingPart(exp2)


if initialSubstitution != []:
    tail1 = apply(tail1, initialSubstitution)
    tail2 = apply(tail2, initialSubstitution)
remainingSubstitution = unify(tail1, tail2)

if not remainingSubstitution:

```

```
    return False

initialSubstitution.extend(remainingSubstitution)
return initialSubstitution
```

```
exp1 = "knows(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

OUTPUT

```
Substitutions:
[('X', 'Richard')]
```

```
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

**9. Convert a given first order logic statement into Conjunctive Normal Form (CNF).**

### 9. Convert FOL to CNF

Step 1: Create a list of SKOLEM CONSTANTS

Step 2: Find

if the attributes are lowercase, replace them with a Skolem constant.

else use Skolem constant as function term list

if the attributes are both lowercase or uppercase replace the respective attributes with a return function.

Step 3: replace  $\leftrightarrow$  with ' $\neg$ '

transform  $\neg \phi \Leftrightarrow \phi = (\phi \Rightarrow \phi_1) \wedge (\phi_1 \Rightarrow \phi)$

Step 4: replace  $\Rightarrow$  with ' $\neg$ '

Step 5: Apply de Morgan law.

replace  $\neg\neg$

or replace it (if was present)

replace  $\neg\neg\neg$

or replace it (if was present)

replace  $\neg\neg\neg\neg$

```
def getAttributes(string):
```

```
    expr = '
```

```
    matches = re.findall(expr, string)
```

```
    return [m for m in str(matches) if m.isalpha()]
```

```
def getPredicates(string):
```

```
    expr = '[a-zA-Z]+
```

```
    return re.findall(expr, string)
```

```

def DeMorgan(sentence):
    string = ".join(list(sentence).copy())
    string = string.replace('~~',")
    flag = '[' in string
    string = string.replace('~[',"")
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ".join(s)
    string = string.replace('~~',")
    return f'{string}' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ".join(list(sentence).copy())
    matches = re.findall('[\forall\exists].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, "")
        statements = re.findall(
            ]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
    for predicate in getPredicates(statement):

```

```

        attributes = getAttributes(predicate)

        if ".join(attributes).islower():

            statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))

        else:

            aL = [a for a in attributes if a.islower()]

            aU = [a for a in attributes if not a.islower()][0]

            statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({{aL[0]} if
len(aL) else match[1]}})')

        return statement

import re

def fol_to_cnf(fol):

    statement = fol.replace("<=>", " ")

    while ' ' in statement:
        i = statement.index(' ')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']&[' + statement[i+1:] +
'=>' + statement[:i] + ']'
        statement = new_statement

    statement = statement.replace("=>", "-")

    expr = '

    statements = re.findall(expr, statement)

    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'

    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

```

```

while '-' in statement:
    i = statement.index('-')
    br = statement.index('[') if '[' in statement else 0
    new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
    statement = statement[:br] + new_statement if br > 0 else new_statement

while '~V' in statement:
    i = statement.index('~V')
    statement = list(statement)
    statement[i], statement[i+1], statement[i+2] = 'V', statement[i+2], '~'
    statement = ".join(statement)

while '~E' in statement:
    i = statement.index('~E')
    s = list(statement)
    s[i], s[i+1], s[i+2] = 'E', s[i+2], '~'
    statement = ".join(s)

statement = statement.replace('~[V],[~V]')
statement = statement.replace('~[E],[~E]')
expr = '(~[V|E].)'
statements = re.findall(expr, statement)

for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
    expr = '~

statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, DeMorgan(s))

return statement

print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))

```

```
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)⇒loves(x,y)]⇒[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]⇒criminal(x)"))
```

OUTPUT

JAGADEESH(1BM21CS079)

Enter FOL statement: x+y\_z\*s  
FOL converted to CNF: [~x+y | z\*s] & [~z\*s | x+y]

10. Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

10

3) Code for KB consistency for prime for given query using Forward chaining.

isogeny

der reziproke ( $\times$ ):

network  $S(x) = \{x \in \mathbb{R}^n \mid f(x) \leq 0\}$ .

at geodetic (Strong)

$$\text{area} = ((a-2T+2)(c-2+T-a))$$

~~when we call (new func)~~

Dear Lucia

Chlorophyll (light absorption)

2017-2018 学年

area of study:  $\text{percentage} = \frac{\text{Actual Area}}{\text{Estimated Area}}$

3.18  $\neg \exists x \phi(x) = \forall x \neg \phi(x)$

*Dark Brown - Brown*

$$3x^4 - 2x^3 + 5x^2 - 2x + 1 = 2x^4(1.5x^2 - 1) + x^2(5x - 2) + 1$$

Chlorophyll-a

44. *lutea* (not *prae-*)

$$g_{\mu\nu} = g_{\mu\mu} + g_{\nu\nu}$$

$\mathbb{R} = \mathbb{C} \cap \mathbb{R}^2$  ( $\mathbb{C} = \mathbb{R}^2$ )

$\text{diff}(f) = \{ f(x+1) - f(x) \mid x \in \mathbb{N}_0 \}$

$$x(t, y_0) = \text{tanh}(\alpha C t)$$

See reduced test bags

## Section A - 17

卷之二

A diagram of a single neuron. It features a central circular cell body labeled "cell body". Two short, branching lines extend from the left side of the cell body, labeled "dendrite". A long, thin line extends from the right side of the cell body, labeled "axon".

For Reg. in customs:  
if customs reg.:

multiple levels (e.g.) at 2-3 (non-linear) (not  
( $\Sigma$ , E, yes/no) + ( ) for  $\delta$  in each)  
else stuck

class XIS

~~def init - (self):~~  
~~def self, self = self()~~  
~~self. implements = self()~~

did well (last, e').

It is  $\Rightarrow$  we  
test implication and implement it?

2nd, 3rd, and 4th (and so on).

then  $\tau$  is left-rightward.  
and evaluate left-hand  
at  $\tau\tilde{\tau}$ .  
such. Right-evaluates

Q. A. Gregory (left):  
Please "tell both!"  
There is, I am convinced, a lot of experience  
there & in research.  
Please check it out.

kb = KB()  
 kb.tell('Ring(x) & greedy(x) => evil(x)')  
 kb.tell('King(John)')  
 kb.tell('greedy(John)')  
 kb.tell('King(Richard)')  
 kb.query('evil(x)')

output:  
Querying  
 1. evil(John)

(Q) King  
 (Q) evil

import re

```
def isVariable(x):
  return len(x) == 1 and x.islower() and x.isalpha()
```

```
def getAttributes(string):
```

```
  expr = '
```

```
  matches = re.findall(expr, string)
```

```
  return matches
```

```

def getPredicates(string):
    expr = '([a-zA-Z]+)[^&|]+'
    return re.findall(expr, string)

class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):

```

```

c = constants.copy()

f = f"{{self.predicate}({{',join([constants.pop(0) if isVariable(p) else p for p in self.params])}})"

return Fact(f)

class Implication:

    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            attributes = attributes.replace(key, constants[key])
        expr = f'{predicate} {attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None

class KB:

```

```

def __init__(self):
    self.facts = set()
    self.implications = set()

def tell(self, e):
    if '=>' in e:
        self.implications.add(Implication(e))
    else:
        self.facts.add(Fact(e))
    for i in self.implications:
        res = i.evaluate(self.facts)
        if res:
            self.facts.add(res)

def query(self, e):
    facts = set([f.expression for f in self.facts])
    i = 1
    print(f'Querying {e}:')
    for f in facts:
        if Fact(f).predicate == Fact(e).predicate:
            print(f'\t{i}. {f}')
            i += 1

def display(self):
    print("All facts: ")
    for i, f in enumerate(set([f.expression for f in self.facts])):
        print(f'\t{i+1}. {f}')

kb = KB()

```

```
kb.tell('missile(x)=>weapon(x)')

kb.tell('missile(M1)')

kb.tell('enemy(x,America)=>hostile(x)')

kb.tell('american(West)')

kb.tell('enemy(Nono,America)')

kb.tell('owns(Nono,M1)')

kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')

kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')

kb.query('criminal(x)')

kb.display()
```

#### OUTPUT

JAGADEESH(1BM21CS079)

```
→ Enter number of statements in Knowledge Base: 4
Elephant(x) => Mammal(x)
Lion(Mufasa)
Mammal(x) => Animal(x)
Animal(Simba)
Enter Query:
Mammal(x)
Querying Mammal(x):
All facts:
1. Lion(Mufasa)
2. Animal(Simba)
```