

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT On

ARTIFICIAL INTELLIGENCE

Submitted by

PALLAVI MANUBALLA (1BM21CS124)

**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 **PALLAVI MANUBALLA(1BM21CS124)**, 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.

Saritha A N
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Table of Contents

SL No	Name of Experiment	Page No
1	Implement Tic –Tac –Toe Game	1-6
2	Implement 8 puzzle problem	7-9
3	Implement Iterative deepening search algorithm.	9-12
4	Implement A* search algorithm.	12-16
5	Implement vaccum cleaner agent.	17-20
6	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .	20-22
7	Create a knowledge base using prepositional logic and prove the given query using resolution	22-28
8	Implement unification in first order logic	28-30
9	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	30-34
10	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	34-38

1.Implement Tic –Tac –Toe Game.

```
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



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



1	2	3
4	5	6
7	8	9

computer's turn :

1	X	3
4	5	6
7	8	9

Your turn :

enter a number on the board :4



Your turn :

enter a number on the board :4



1	X	3
0	5	6
7	8	9

computer's turn :

X	X	3
0	5	6
7	8	9

Your turn :

enter a number on the board :5



```
+-----+
```

Your turn :



enter a number on the board :5

```
+-----+
```

X	X	3
---	---	---

```
+-----+
```

0	0	6
---	---	---

```
+-----+
```

7	8	9
---	---	---

```
+-----+
```

computer's turn :

```
+-----+
```

X	X	X
---	---	---

```
+-----+
```

0	0	6
---	---	---

```
+-----+
```

7	8	9
---	---	---

```
+-----+
```

winner is X

2 . Solve 8 puzzle problems.

```
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



```
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.

```
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')
```

```
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]]
```

4. Implement A* search algorithm.

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 contains position values for moving the blank space in either of
```

```
        the 4 directions [up,down,left,right] respectively. """
```

```
    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

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

Enter the start state matrix



```
1 2 3
4 5 6
_ 7 8
```

Enter the goal state matrix

```
1 2 3
4 5 6
7 8 _
```

```
|
|
|
\'/
```

```
1 2 3
4 5 6
_ 7 8
```

```
|
|
|
\'/
```

```
1 2 3
4 5 6
7 _ 8
```

```
|
```

```
-
```

```
|
|
|
\'/
```

```
1 2 3
4 5 6
7 8 _
```

5. Implement vaccum cleaner agent.

```
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:

```
— 0 indicates clean and 1 indicates dirty
Enter Location of Vacuum b
Enter status of b1
Enter status of other room1
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 LEFT 2
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 .

```
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:

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

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

```
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}.\t| {step}\t| {steps[step]}\t')
        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:

```
[ '~P', 'R' ]
```

```
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. \
            \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. \
                \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.	$R \vee \neg P$	Given.
2.	$R \vee \neg Q$	Given.
3.	$\neg R \vee P$	Given.
4.	$\neg R \vee Q$	Given.
5.	$\neg R$	Negated conclusion.
6.		Resolved $R \vee \neg P$ and $\neg R \vee P$ to $R \vee \neg R$, which is in turn null.

A contradiction is found when $\neg R$ 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.	$P \vee Q$	Given.
2.	$\neg P \vee R$	Given.
3.	$\neg Q \vee R$	Given.
4.	$\neg R$	Negated conclusion.
5.	$Q \vee R$	Resolved from $P \vee Q$ and $\neg P \vee R$.
6.	$P \vee R$	Resolved from $P \vee Q$ and $\neg Q \vee R$.
7.	$\neg P$	Resolved from $\neg P \vee R$ and $\neg R$.
8.	$\neg Q$	Resolved from $\neg Q \vee R$ and $\neg R$.
9.	Q	Resolved from $\neg R$ and $Q \vee R$.
10.	P	Resolved from $\neg R$ and $P \vee R$.
11.	R	Resolved from $Q \vee R$ and $\neg Q$.
12.		Resolved R and $\neg R$ to $R \vee \neg R$, which is in turn null.

A contradiction is found when $\neg R$ is assumed as true. Hence, R is true.

8. Implement unification in first order logic

import re

```
def getAttributes(expression):
```

```
    expression = expression.split("(")[1:]
```

```
    expression = "(" + ".join(expression)
```

```
    expression = expression[:-1]
```

```
    expression = re.split("(?"
```

```
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).

```
def getAttributes(string):  
    expr = '  
,  
    matches = re.findall(expr, string)  
    return [m for m in str(matches) if m.isalpha()]  
  
def getPredicates(string):  
    expr = '[a-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('[∀∃].', 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 '~∀' in statement:
    i = statement.index('~∀')
    statement = list(statement)
    statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
    statement = ''.join(statement)

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

statement = statement.replace('~[∀', '[~∀')
statement = statement.replace('~[∃', '[~∃')

expr = '(~[∀|∃]).'

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

```

[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))][loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)

```

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

```
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-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:
        if constants[key]:
            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

```
Querying criminal(x):
```

```
    1. criminal(West)
```

```
All facts:
```

```
    1. enemy(Nono,America)
```

```
    2. hostile(Nono)
```

```
    3. sells(West,M1,Nono)
```

```
    4. criminal(West)
```

```
    5. owns(Nono,M1)
```

```
    6. weapon(M1)
```

```
    7. american(West)
```

```
    8. missile(M1)
```