VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence
(22CS5PCAIN)

Submitted by

Varun Raj S (1BM21CS264)

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
Nov-2023 to 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 **Varun Raj S (1BM21CS264),** 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 academic semester Nov-2023 to Feb-2024. The Lab report has been approved as it satisfies the academic requirements in respect of **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Prof. Swathi Sridharan

Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

Lab Program No.	Program Details	Page No.
1	Written lab observation pictures.	-
2	Implement Tic –Tac –Toe Game.	1 - 6
3	Solve 8 puzzle problems.	7 - 10
4	Implement Iterative deepening search algorithm.	11 - 14
5	Implement A* search algorithm.	15 - 19
6	Implement vaccum cleaner agent.	20 - 22
7	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not.	23 - 24
8	Create a knowledge base using prepositional logic and prove the given query using resolution	25 - 29
9	Implement unification in first order logic	30 - 35
10	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	36 - 37
11	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	38 - 42

Course Outcome

CO1	Apply knowledge of agent architecture, searching and reasoning techniques for different applications.
CO2	Analyse Searching and Inferencing Techniques.
CO3	Design a reasoning system for a given requirement.
CO4	Conduct practical experiments for demonstrating agents, searching and inferencing.

Lab observation:

Date __/__/__
Page _____

Experiment-1 : Implement Tic Tac Toe

Tic Tac Toe is a two-player, zero-sum game. We use the minimax algorithm to implement the game of tic tac toe.

Minimax Algorithm

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player.

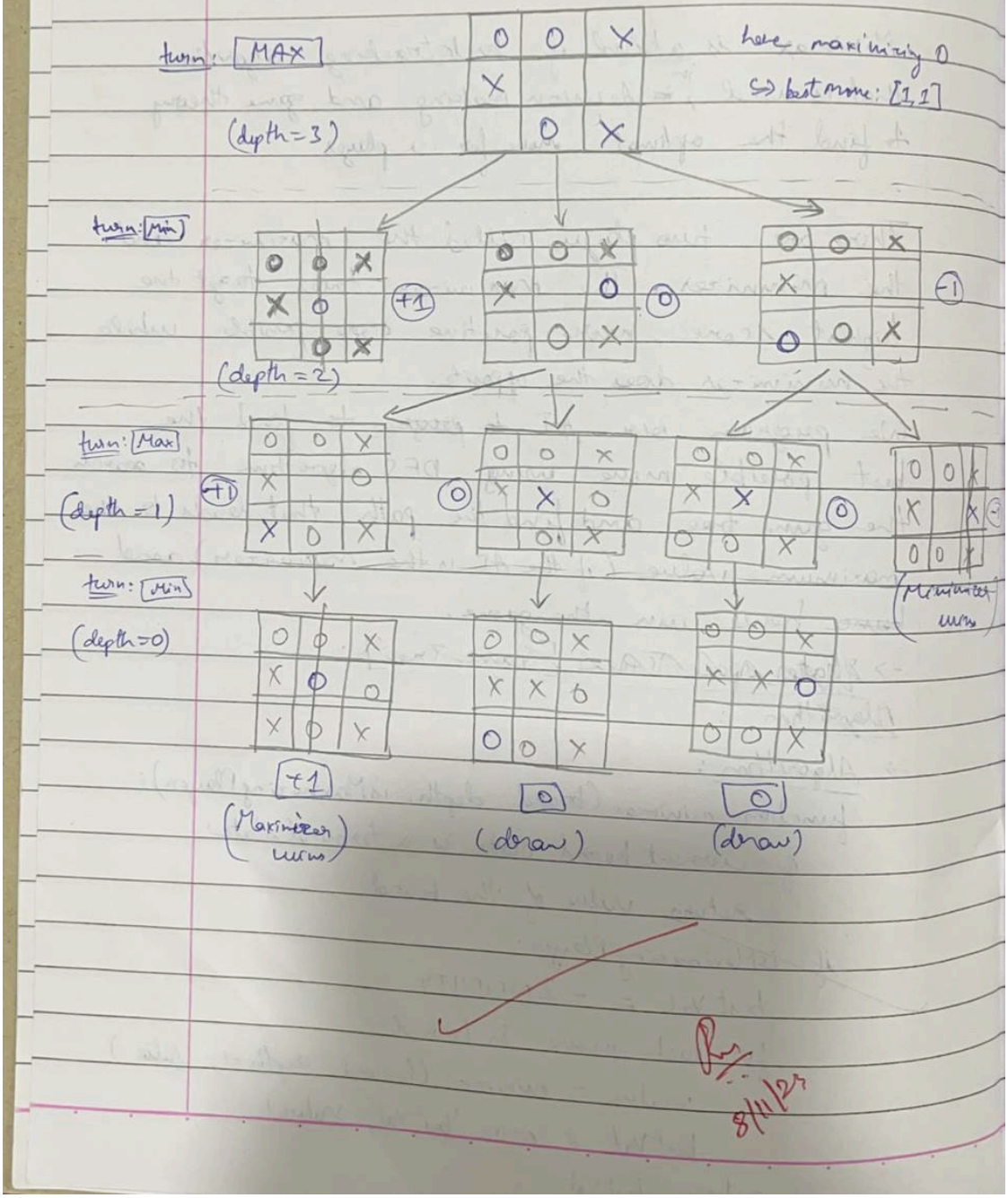
There are two players, called the maximizer and the minimizer. The maximizer tries to get the highest score, more positive score possible, while the minimizer does the opposite.
We program our AI to find the best possible move using DFS algorithm to search the game tree and find the path that leads to the maximum value (if the AI is the maximizer) and hence win the game.

→ Algorithm:

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state:
        return value of the board
    if isMaximizingPlayer:
        bestVal = -INFINITY
        for each move in board:
            value = minimax(board, depth+1, false)
            bestVal = max(bestVal, value)
    return bestVal
```

else:
    bestVal = +INFINITY
    for each move in board:
        value = minimax(board, depth+1, true)
        bestVal = min(bestVal, value)
    return bestVal.

→ Game Tree [state space tree]:



8/11/23

→ Code:

```
import math
import copy

X = "X"
O = "O"
EMPTY = None

def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO = 0
    countX = 0
    for y in [0,1,2]:
        for x in board[y]:
            if x == "O":
                countO = countO + 1
            elif x == "X":
                countX += 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O

def actions(board):
    freeboxes = set()
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board[i][j] == EMPTY:
                freeboxes.add((i,j))
    return freeboxes
```

```
def result(board, action):
    i = action[0]
    j = option[1]
    if type(action) == list:
        action = (i,j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X
        elif player(board) == O:
            board[i][j] = O
    return board

def winner(board):
    if (board[0][0] == board[0][1] == board[0][2] == X
        or board[1][0] == board[1][1] == board[1][2] == X
        or board[2][0] == board[2][1] == board[2][2] == X):
        return X

    if (board[0][0] == board[0][1] == board[0][2] == O or
        board[1][0] == board[1][1] == board[1][2] == O or
        board[2][0] == board[2][1] == board[2][2] == O):

    for i in [0,1,2]:
        s2 = []
        for j in [0,1,2]:
            s2.append(board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]

    strikeD = []
    for i in [0,1,2]:
        strikeD.append(board[i][i])
```

```
if (striked[0] == striked[1] == striked[2]):
    return striked[0]
if (board[0][2] == board[1][1] == board[2][0]):
    return board[0][2]
return none

def terminal(board):
    Full = True,
    for i in [0,1,2]:
        for j in board[i]:
            if j is None:
                Full = False
if Full:
    return True
if (Winner(board) in not None):
    return True
    return False

def utility(board):
    if (winner(board) == 'X'):
        return 1
    elif (winner(board) == 'O':
        return -1
    else:
        return 0

def minimax_helper(board):
    isMaxTurn = True
    if player(board) == X else Fals
    if terminal(board):
        return utility(board)
    scores = []
    for move in action
```

```
    for move in actions(board):
        result(board, move)
        scores.append(minimax_helper(board))
        board[move(0)].[move(1)] = EMPTY
    root
    return max(scores) if isMaxTerm else min(score)
def minimax(board):
    isMaxTerm = True if player(board) == X else false
    bestScore = None
    if isMaxTerm:
        bestScore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move
        return bestMove
else else:
    bestScore = +math.inf
    for move in actions(board):
        result(board, move)
        score = minimax_helper(board)
        board[move[0]][move[1]] = EMPTY
        if (score is bestScore):
            bestScore = score
            bestMove = move
    return bestMove
```

```
def print_board(board):
    for row in board:
        print(row)
game_board = initial_state()
print("Initial board:")
print_board(game_board)

while not terminal(game_board):
    if player(gameboard) == X:
        user_input = input("Enter your move
                            (row column):")
    row, colum = map(int user_input.split(','))
    result(game_board,(row, col))
else
    print("AI is making a move"),
    move = minimax(copy.decopy(game_board))
    result(game_board, move)

print("current Board")
move = minimax(copy.decopy(game_board)
result(gameboard, move)

if winner(game_board) is ~~none~~ not none:
    print("The winner is: {winner(gameboard)}")
else:
    print("It is a tie")
```

Page

Program - 2 : ~~8-Puzzle Problem~~ ~~Vaccum Cleaner Agent~~

```
def bfs( src, target ):
    queue = []
    queue.append (src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp. append (source)

        print(source)

        if source == target:
            print ("success")
            return
        poss_moves_to_do = []
        poss_moves_todo = possible_moves (source, exp)

        for move in poss_moves_todo:

            if move not in exp and move not in queue:
                queue. append(move)

def possible_moves (state, visited_states

    b = state. index(0)

    d = []
```

```
if b not in [0,1,2]:
    d.append('u')
if b not in [6,7,8]:
    d.append('d')
if d 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 = [1,2,3,0,4,5,6,0,7,8]
target = [1,2,3,4,5,0,6,7,8]

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

bfs(src, target)

Output:

```
def iddfs(src, target, depth):
    for i in range(depth):
        visited_states = []
        if dfs(src, target, i+1, visited_states):
            return True
    return False
```

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

```
result = bfs(src, target)
print("Solution:", result)
```

Date __/__/__
Page _____

Program-3: Vaccum Cleaner Agent

```
def vacuum_world():

    goal_state = {'A': '0': 'B': '0'}
    cost = 0

location_input = input("Enter location of Vocuum")

status_input = input("Enter status of " + location input)

status_input = input("Enter status of other room")
print("Initial Location Condition" + str(goal_state))

if location_input == 'A':

    print("Vacuum is placed in Location A")
    if status_input == '1':
        print("Location A is dirty.")

        goal_state['A'] = '0'
        cost += 1
        print("Cost for CLEANING A" + str(cost))
        print("Location A has been changed.")
    else:
        print("No action" + str(cost))
        print("Location B is already clean.")

if status_input == '0':
    print("Location A is already clean")
    if status_input_complement == '1':
        print("Location B is dirty")
```

```
print("Moving RIGHT to the Location B")
Cost += 1
print("COST for moving RIGHT "
      + str(cost))
goal_state['B'] = '0'
cost += 1
print("Cost for SUCK" + str(cost))
print("Location B has been cleaned.")
else
else:
    print("No action" + str(cost))
    print(cost)
    print("Location B is already clean.")

else:
    print("Vaccum is placed in location B")

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

        goal_state['B'] = '0'
        Cost += 1
        print("COST of for CLEANING " + str(cost))
        print("Location B has been Cleaned")

    if status input complement == '1':
        print("Location A is dirty")
        print("Moving LEFT to the location A.")
        cost += 1
        print("COST for moving LEFT" + str(cost))
        goalstate['A'] = '0'
        cost += 1
        print("cost for SUCK" + str(cost))
        print("Location A has been Cleaned.")
```

```
else:
    print(cost)
    print("Location B is already clean")
    if status_input_complement == '1':
        print("Location A is Dirty.")
        print("Moving LEFT to the location A.")
        cost += 1
        print("COST for moving LEFT " + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("Cost for SUCK " + str(cost))
        print("Location A has been cleaned")
    else:
        print("No action " + str(cost))
        print("Location A is already clean.")
print("GOAL STATE:")
print(goal_state)
print("Performance Measurement: " + str(cost))

vacuum_world()
```

Program-4: 8-puzzle using iterative deeping search Algorithm

Test case:

source state:

1	2	3
-1	4	5
6	7	8

target:

1	2	3
4	5	-1
6	7	8

depth = 2

Output: True. True.

→ Code:

```
def dfs(src, target, limit, visited):
    if src == target:
        return True
    if limit <= 0:
        return False
    visited.append(src)
    moves = possible moves(src, visited)
    for move in moves:
        if dfs(move, target, limit-1, visited):
            return True
    return False

def possible moves(state, visited):
    b = state.index(-1)
    d = []
```

```
if b not in [0,1,2]:
    d += 'u'

if b not in [6,7,8]:
    d += 'd'
if b not in [2,5,8]:
    d += 'r'
if b not in [0,3,6]:
    d += 'l'

pos_moves = []
for move in d:
    pos_moves.append(gen(state, move, b))
return [move for move in pos_moves if move
        not in visited]

def gen(state, move, blank):
    temp = state.copy()

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

    return temp.
```

```
def iddfs(src, target, depth):
    for i in range(depth):
        visited_states = []
        if dfs(src, target, i+1, visited):
            return True
    return False

src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]

depth = 2
result = iddfs(src, target, depth)
print("solution", result)
```

5) A* Algorithm

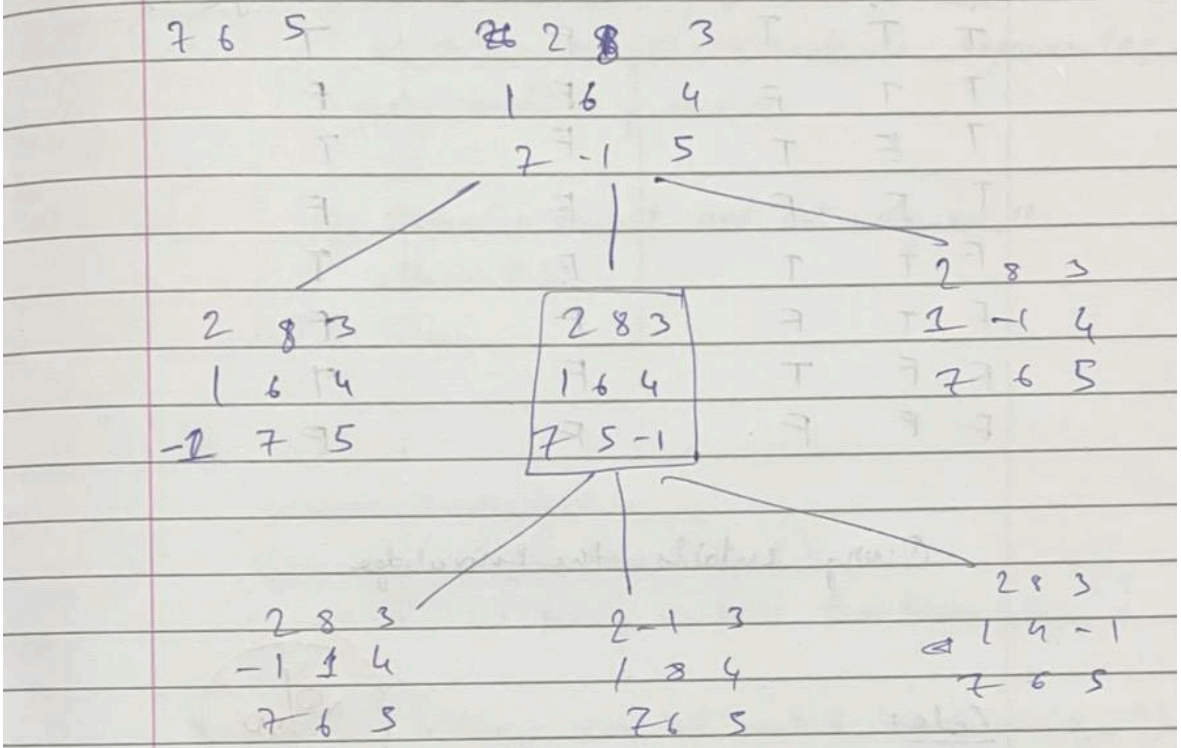
```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(f"""
    {state[0]} {state[1]} {state[2]}
    {state[3]} {state[4]} {state[5]}
    {state[6]} {state[7]} {state[8]}
    """)

def h(state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target[j]:
            count = count + 1
        return count

def astar(state, target):
    states = [src]
    g = 0
    visited_state = []
    while len(state):
        print(f"level {g} ").
        moves = []
        for state in states:
            visited_states.append(state)
        print(state)
        if state == target:
            print("success")
            return
```

```
moves += [move for move in possible_moves
          (state, visited_states) if move not in
          moves]
costs = [g+h(move, target) for move in moves]
states = [moves[i] for i in range(len(moves)) if
          costs[i] == min(costs)]
g += 1
print("failed")

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if b is in range(6):
        d.append('u')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')
    if b in range(3):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if
            move not in visited_states]
def gen(state, m, b):
    temp = state.copy()
    if m == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if m == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
```

Date __/__/__
Page _____

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

goal:
1 2 3
8 -1 4
7 6 5



[(Not q) or (not p) & r] and [(not p) or q]
~q ~ ¬ Date Page

6) Knowledge base using Propositional logic

→ ~~Knowledge~~ Expression: (~q v ~p v r) ^ (~q ^ p) ^ q

query : r.

p	q	r	KB	Query (r)
T	T	T	F	T
T	T	F	F	F
T	F	T	F	T
T	F	F	F	F
F	T	T	F	T
F	T	F	F	F
F	F	T	F	T
F	F	F	F	F

Query entails the knowledge.

10/10

Code:

```
def evaluate_expression(p, q, r):
    expression_result = (p or q) and (not q or p)
    return expression_result

def generate_truth_table():
    print("p | q | r | Expression(KB) | Query (p ^ r)")
    print("--- | --- | --- | --- | ---")

    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r
```

20/12/23

Date __/__/____
Page _____

```
print(f"{p}|{q}|{r}|{expression_result}|{query_result}")

def query_entails_knowledge():
    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p,q,r)
                query_result = p and r.

                if expression_result and not query_result:
                    return False
    return True

def main():
    generate_truth_table()
    if query_entails_knowledge():
        print("\n Query entails the knowledge")
    else:
        print("\n Query does not entail the knowledge.")

main()
```

Date __/__/__
Page _____

7) KB using propositional logic and prove the given query using resolution.

(i) KB: R v ~P R v ~Q ~R v P ~R v Q

query : R

Step	Clause	Derivation
1	R v ~P	Given.
2	R v ~Q	Given
3	~R v Q	Given
4	~R v P	Given
5	~R	Negated conclusion
6	~~R~~ R v ~R	~~4 on~~ 2 and 3

Therefore null clause got.

Contradiction is found when ~R (negated query) taken. Thus R is true.

Program:

```
from z3 import Implies, Not, Bool, Solver, sat

p,q,r = Bool('p'), Bool('q'), Bool('r')

kb = Implies(p,q) & Implies(q,r) & Not(p)

query = r

def prove_query_with_resolution(knowledge_base, query):
    s = solver()
    s.add(Not(knowledge_base), query)
    result = s.check()
    return result == sat
```

```
proof_result = prove_query_with_resolution(kb, query)

if proof_result:
    print("The query is proved to be true")
else:
    print("The query is not proved to be true")
```

Date __/__/__
Page _____

8) Unification of first Order logic

Enter the first Expression : Knows(f(x), y)
Enter the second Expression : Knows(J, John)
The substitutions are:
['g / f(x)', 'John / y']

Code:

```
def unify(expr1, expr2):

    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

if func1 != func2:
    print("Different functions, cannot unify")
    return None.

args1 = args1.rstrip(')').split(',')
args2 = args2.rstrip(')').split(',')

substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower() and a1 != a2:
            substitution[a1] = a2
        elif a1.islower() and not a2.islower():
            substitution[a1] = a2
    elif not a1.islower() and a2.islower():
            substitution[a2] = a1
    elif a1 != a2:
        print("Expressions cannot be unified")
        return None
return substitution
```

```
def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return expr

expr1 = input("Enter first expression: ")
expr2 = input("Enter second expression: ")

substitution = unify(expr1, expr2)

if substitution:
    print("The substitutions are:")
    out = []
    for key, value in substitution.items():
        out += [key + " / " + value]
    print(out)

expr1_result = apply_substitution(expr1, substitution)
expr2_result = apply_substitution(expr2, substitution)

print(f'Unified expression 1: {expr1_result}')
print(f'Unified expression 2: {expr2_result}')
```

Vol 1

Date __/__/____

Page _____

Convert given first Order logic statement into Conjunctive Normal Form (CNF)

Input : $\forall x \cdot food(x) \Rightarrow likes(John, x)$

Step-1 : Eliminate $\Rightarrow$ and rewrite:

$\forall x \neg food(x) \vee likes(John, x)$

Step-2 : Move negation "$\neg$" inwards.

&

Step-3 : Standardize variables apart by renaming them.

Step-4 : Skolemize each existential variable is replaced by a skolem constant or Skolem function of the enclosing universally quantified variable.

$\forall x \neg food(A) \vee likes(John, A)$

, where A is the new skolem ~~function~~ constant.

Step-5 : Drop universal quantifiers :

$\neg food(A) \vee likes(John, A)$

Step-6 : De Morgan's ~~law~~ application :

(Here we don't need to use De Morgans law)

Date / /
Page

Example-2 : $\forall x [\exists z [loves(x,z)]]$

→ No "⇒" or negation in FOL statement, so continue to step 4

→ Drop universal quantifiers

Op: ~~$[loves(x, B(x))]$~~ new skolem function B(x)

→ No

$\forall x [\exists z [loves(x,z)]]$

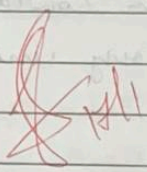
→ Step-4 : Skolemize: $z = B(x)$

new skolem function: B.

$\forall x [\exists z [loves(x, B(x))]]$

→ Step-5 : Drop universal quantifiers:

$[loves(x, B(x))]$

→ 

→ Code:

```
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~]+\([A-Za-z,]+\)'
    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('[∀∃].', 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

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 //( inside while block
        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] + 'V' +
            statement[i+1:]

    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], '~'
    statent = ''.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))
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement.

inp_str = input("Enter the FOL statement:")
print(Skolemization(fol_to_cnf(inp_str)))
```

Date __/__/__
Page

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

Output:

(1) Querying evil(x):
1. evil(John)

All facts:
1. evil(John)
2. greedy(John)
3. king(John)
4. king(Richard)

(2) Querying criminal(x):
1. criminal(west)

All facts:
1. hostile(nano)
2. criminal(West)
3. owns(Nano, M1)
4. Missile(M1)
5. American(west)
6. Weapon(M1)
7. Sells(West, M1, Nano)
8. Enemy(nano, America)

FOL: Missile(x) ⇒ weapon(x)
missile(M1)
enemy(x, America) ⇒ hostile(x)
america(west)
enemy(nano, America)
owns(Nano, M1)

Date __/__/__

Page _____

american (west)
enemy (Nano, America)
owns (Nano, ML)
missile (x) & owns (nano, x) ⇒ sells (west, x, nano)

hostile (z) ⇒ criminal (x)
Criminal (x)

FOL: King (x) & greedy (x) ⇒ evil (x)
King (John)
greedy (John)
King (Richard)
evil (x)

Code:

```
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.splitExpression(~~
        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 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,America')
kb.tell('American(x) & weapon(y) & sells(x,y,z) &
                    hostile(z) => criminal(x)')
kb.query('criminal(x)')
kb.display()
```

Scroll

1. **Implement Tic –Tac –Toe Game.**

```
import math
import copy

X = "X"
O = "O"
EMPTY = None

def initial_state():
    return [[EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY],
            [EMPTY, EMPTY, EMPTY]]

def player(board):
    countO = 0
    countX = 0
    for y in [0, 1, 2]:
        for x in board[y]:
            if x == "O":
                countO = countO + 1
            elif x == "X":
                countX = countX + 1
    if countO >= countX:
        return X
    elif countX > countO:
        return O

def actions(board):
    freeboxes = set()
```

```
    for i in [0, 1, 2]:
        for j in [0, 1, 2]:
            if board[i][j] == EMPTY:
                freeboxes.add((i, j))
    return freeboxes


def result(board, action):
    i = action[0]
    j = action[1]
    if type(action) == list:
        action = (i, j)
    if action in actions(board):
        if player(board) == X:
            board[i][j] = X
        elif player(board) == O:
            board[i][j] = O
    return board


def winner(board):
    if (board[0][0] == board[0][1] == board[0][2] == X or board[1][0] == board[1][1] ==
board[1][2] == X or board[2][0] == board[2][1] == board[2][2] == X):
        return X
    if (board[0][0] == board[0][1] == board[0][2] == O or board[1][0] == board[1][1] ==
board[1][2] == O or board[2][0] == board[2][1] == board[2][2] == O):
        return O
    for i in [0, 1, 2]:
        s2 = []
        for j in [0, 1, 2]:
            s2.append(board[j][i])
```

```
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]
    strikeD = []
    for i in [0, 1, 2]:
        strikeD.append(board[i][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        return strikeD[0]
    if (board[0][2] == board[1][1] == board[2][0]):
        return board[0][2]
    return None


def terminal(board):
    Full = True
    for i in [0, 1, 2]:
        for j in board[i]:
            if j is None:
                Full = False
    if Full:
        return True
    if (winner(board) is not None):
        return True
    return False


def utility(board):
    if (winner(board) == X):
        return 1
    elif winner(board) == O:
        return -1
```

```
    else:
        return 0


def minimax_helper(board):
    isMaxTurn = True if player(board) == X else False
    if terminal(board):
        return utility(board)

    scores = []
    for move in actions(board):
        result(board, move)
        scores.append(minimax_helper(board))
        board[move[0]][move[1]] = EMPTY
    return max(scores) if isMaxTurn else min(scores)


def minimax(board):
    isMaxTurn = True if player(board) == X else False
    bestMove = None
    if isMaxTurn:
        bestScore = -math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score > bestScore):
                bestScore = score
                bestMove = move
        return bestMove
```

```
    else:
        bestScore = +math.inf
        for move in actions(board):
            result(board, move)
            score = minimax_helper(board)
            board[move[0]][move[1]] = EMPTY
            if (score < bestScore):
                bestScore = score
                bestMove = move
        return bestMove


def print_board(board):
    for row in board:
        print(row)


# Example usage:
game_board = initial_state()
print("Initial Board:")
print_board(game_board)

while not terminal(game_board):
    if player(game_board) == X:
        user_input = input("\nEnter your move (row, column): ")
        row, col = map(int, user_input.split(','))
        result(game_board, (row, col))
    else:
        print("\nAI is making a move...")
        move = minimax(copy.deepcopy(game_board))
```

```
        result(game_board, move)


    print("\nCurrent Board:")
    print_board(game_board)


# Determine the winner
if winner(game_board) is not None:
    print(f"\nThe winner is: {winner(game_board)}")
else:
    print("\nIt's a tie!")
```

OUTPUT:

```
Initial Board:
[None, None, None]
[None, None, None]
[None, None, None]

Enter your move (row, column): 1,2

Current Board:
[None, None, None]
[None, None, 'X']
[None, None, None]

AI is making a move...

Current Board:
[None, None, None]
[None, 'O', 'X']
[None, None, None]

Enter your move (row, column): 0,0

Current Board:
['X', None, None]
[None, 'O', 'X']
[None, None, None]

AI is making a move...

Current Board:
['X', 'O', None]
[None, 'O', 'X']
[None, None, None]

Enter your move (row, column): 2,1
```

```
Current Board:
['X', 'O', None]
[None, 'O', 'X']
[None, 'X', None]

AI is making a move...

Current Board:
['X', 'O', None]
[None, 'O', 'X']
['O', 'X', None]

Enter your move (row, column): 1,0

Current Board:
['X', 'O', None]
['X', 'O', 'X']
['O', 'X', None]

AI is making a move...

Current Board:
['X', 'O', 'O']
['X', 'O', 'X']
['O', 'X', None]

The winner is: O
```

2. **Solve 8 puzzle problems.**

```
def bfs(src,target):
    queue = []
    queue.append(src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print(source)

        if source==target:
            print("Success")
            return

        poss_moves_to_do = []
        poss_moves_to_do = possible_moves(source,exp)

        for move in poss_moves_to_do:

            if move not in exp and move not in queue:
                queue.append(move)
def possible_moves(state,visited_states):
    #index of empty spot
    b = state.index(0)

    #directions array
    d = []
```

```
    #Add all the possible directions

    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')



    # If direction is possible then add state to move
    pos_moves_it_can = []

    # for all possible directions find the state if that move is played
    ### Jump to gen function to generate all possible moves in the given directions

    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 new state with tested move to later check if "src == target"
    return temp

print("Example 1")
src= [2,0,3,1,8,4,7,6,5]
target=[1,2,3,8,0,4,7,6,5]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)

print("\nExample 2")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
bfs(src, target)
```

OUTPUT:

```
Example 1
Source:  [2, 0, 3, 1, 8, 4, 7, 6, 5]
Goal State:  [1, 2, 3, 8, 0, 4, 7, 6, 5]
[2, 0, 3, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, 0, 4, 7, 6, 5]
[0, 2, 3, 1, 8, 4, 7, 6, 5]
[2, 3, 0, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, 6, 4, 7, 0, 5]
[2, 8, 3, 0, 1, 4, 7, 6, 5]
[2, 8, 3, 1, 4, 0, 7, 6, 5]
[1, 2, 3, 0, 8, 4, 7, 6, 5]
[2, 3, 4, 1, 8, 0, 7, 6, 5]
[2, 8, 3, 1, 6, 4, 0, 7, 5]
[2, 8, 3, 1, 6, 4, 7, 5, 0]
[0, 8, 3, 2, 1, 4, 7, 6, 5]
[2, 8, 3, 7, 1, 4, 0, 6, 5]
[2, 8, 0, 1, 4, 3, 7, 6, 5]
[2, 8, 3, 1, 4, 5, 7, 6, 0]
[1, 2, 3, 7, 8, 4, 0, 6, 5]
[1, 2, 3, 8, 0, 4, 7, 6, 5]
Success

Example 2
Source:  [1, 2, 3, 0, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 4, 5, 0, 6, 7, 8]
[1, 2, 3, 0, 4, 5, 6, 7, 8]
[0, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 0, 7, 8]
[1, 2, 3, 4, 0, 5, 6, 7, 8]
[2, 0, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, 0, 8]
[1, 0, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, 0, 8]
[1, 2, 3, 4, 5, 0, 6, 7, 8]
Success
```

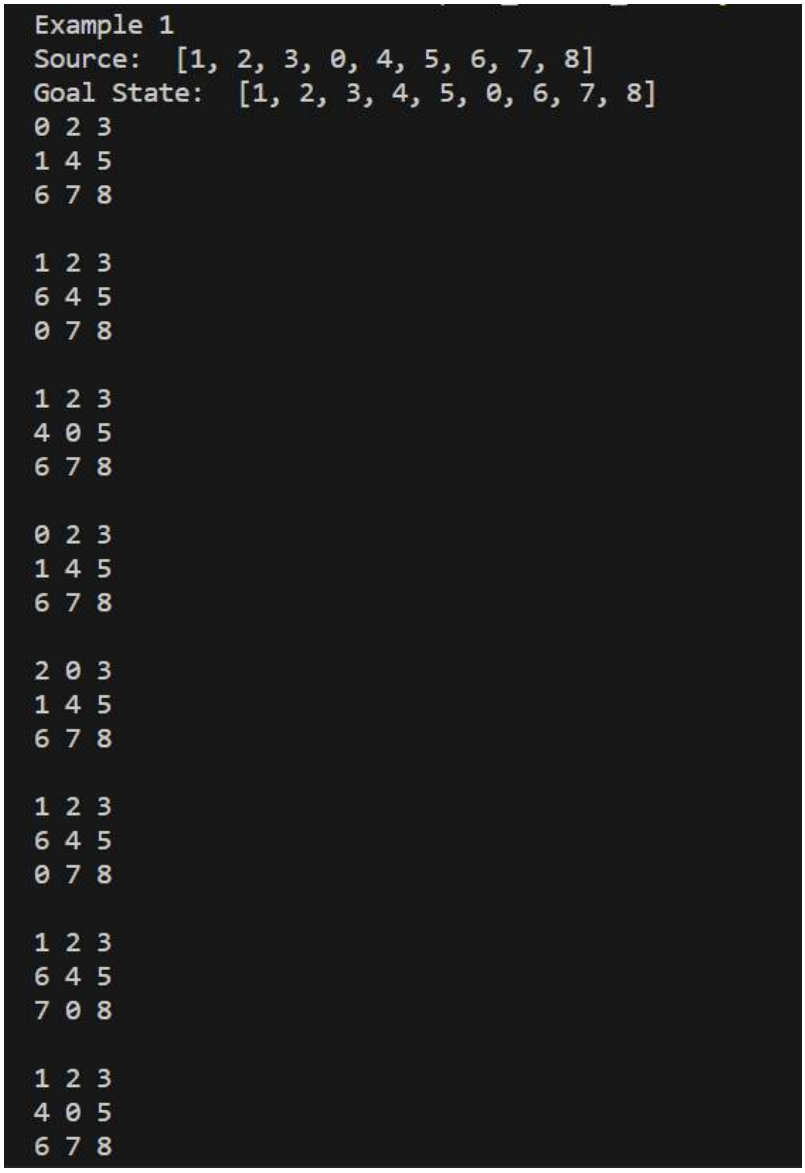
3. **Implement Iterative deepening search algorithm.**

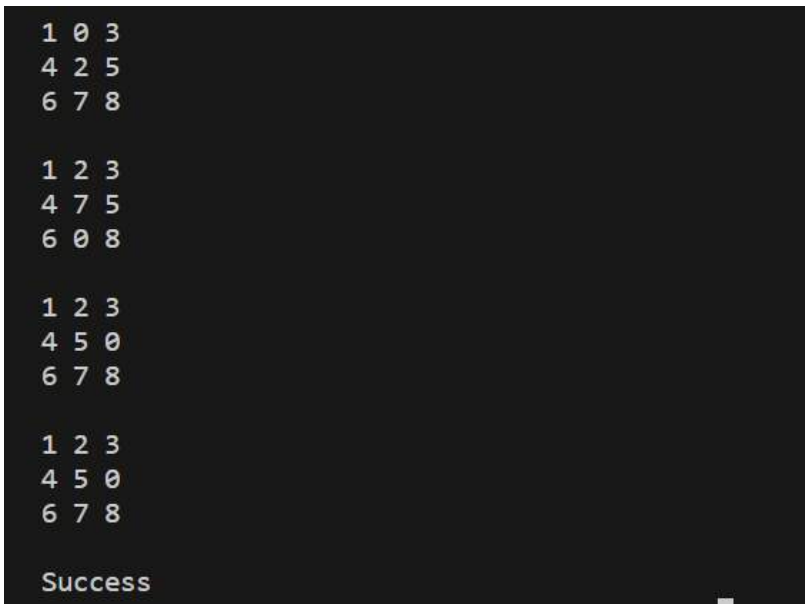
```
def iterative_deepening_search(src, target):
    depth_limit = 0
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is not None:
            print("Success")
            return
        depth_limit += 1
        if depth_limit > 30:  # Set a reasonable depth limit to avoid an infinite loop
            print("Solution not found within depth limit.")
            return

def depth_limited_search(src, target, depth_limit, visited_states):
    if src == target:
        print_state(src)
        return src

    if depth_limit == 0:
        return None

    visited_states.append(src)
    poss_moves_to_do = possible_moves(src, visited_states)

    for move in poss_moves_to_do:
        if move not in visited_states:
            print_state(move)
            result = depth_limited_search(move, target, depth_limit - 1, visited_states)
            if result is not None:
                return result
```

```
    return None


def possible_moves(state, visited_states):
    b = state.index(0)
    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]
    elif m == 'u':
        temp[b - 3], temp[b] = temp[b], temp[b - 3]
```

```
    elif m == 'l':
        temp[b - 1], temp[b] = temp[b], temp[b - 1]
    elif m == 'r':
        temp[b + 1], temp[b] = temp[b], temp[b + 1]

    return temp

def print_state(state):
    print(f"{state[0]} {state[1]} {state[2]}\n{state[3]} {state[4]} {state[5]}\n{state[6]} {state[7]} {state[8]}\n")

print("Example 1")
src = [1,2,3,0,4,5,6,7,8]
target = [1,2,3,4,5,0,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
iterative_deepening_search(src, target)
```

OUTPUT:





4. **Implement A* search algorithm.**

```
def print_grid(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
        f"""
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]} {state[8]}
        """
    )

def h(state, target):
    #Manhattan distance
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3
        dist += abs(x1-x2) + abs(y1-y2)
    return dist

def astar(src, target):
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        moves = []
        for state in states:
            visited_states.add(tuple(state))
```

```
            print_grid(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(state, visited_states) if move not in
moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i] for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")


def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []
    if 9 > b - 3 >= 0:
        d += 'u'
    if 9 > b + 3 >= 0:
        d += 'd'
    if b not in [2,5,8]:
        d += 'r'
    if b not in [0,3,6]:
        d += 'l'
    pos_moves = []
    for move in d:
        pos_moves.append(gen(state,move,b))
    return [move for move in pos_moves if tuple(move) not in visited_states]


def gen(state, direction, b):
    temp = state.copy()
    if direction == 'u':
        temp[b-3], temp[b] = temp[b], temp[b-3]
```

```
    if direction == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if direction == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    if direction == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    return temp

#Test 1
print("Example 1")
src = [1,2,3,-1,4,5,6,7,8]
target = [1,2,3,4,5,-1,6,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 2
print("Example 2")
src = [1,2,3,-1,4,5,6,7,8]
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)


# Test 3
print("Example 3")
src = [1,2,3,7,4,5,6,-1,8]
target=[1,2,3,6,4,5,-1,7,8]
```

```
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)
```

OUTPUT:

```
Example 1
Source:  [1, 2, 3, -1, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 4, 5, -1, 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

Success
Example 2
Source:  [1, 2, 3, -1, 4, 5, 6, 7, 8]
Goal State:  [1, 2, 3, 6, 4, 5, -1, 7, 8]

1 2 3
  4 5
6 7 8


1 2 3
6 4 5
  7 8

Success
```

```
Example 3
Source:  [1, 2, 3, 7, 4, 5, 6, -1, 8]
Goal State:  [1, 2, 3, 6, 4, 5, -1, 7, 8]

1 2 3
7 4 5
6   8


1 2 3
7 4 5
  6 8


1 2 3
  4 5
7 6 8


  2 3
1 4 5
7 6 8


1 2 3
4   5
7 6 8


1 2 3
4 6 5
7   8
```

```
1 2 3
  6 5
4 7 8


1 2 3
6   5
4 7 8


1 2 3
6 7 5
4   8


1 2 3
6 7 5
  4 8


1 2 3
  7 5
6 4 8


  2 3
1 7 5
6 4 8


1 2 3
7   5
6 4 8
```

```
7 1 3
4 6 5
  2 8


7 1 3
4 6 5
2   8


7 1 3
4   5
2 6 8


7 1 3
4 6 5
2 8


7 1 3
  4 5
2 6 8


7 1 3
2 4 5
  6 8

Fail
```

5. Implement vacuum cleaner agent.

```
def clean(floor, row, col):
    i, j, m, n = row, col, len(floor), len(floor[0])
    goRight = goDown = True
    cleaned = [not any(f) for f in floor]
    while not all(cleaned):
        while any(floor[i]):
            print_floor(floor, i, j)
            if floor[i][j]:
                floor[i][j] = 0
                print_floor(floor, i, j)
            if not any(floor[i]):
                cleaned[i] = True
                break
            if j == n - 1:
                j -= 1
                goRight = False
            elif j == 0:
                j += 1
                goRight = True
            else:
                j += 1 if goRight else -1
        if all(cleaned):
            break
        if i == m - 1:
            i -= 1
            goDown = False
        elif i == 0:
            i += 1
```

```
            goDown = True
        else:
            i += 1 if goDown else -1
        if cleaned[i]:
            print_floor(floor, i, j)

def print_floor(floor, row, col): # row, col represent the current vacuum cleaner position
    for r in range(len(floor)):
        for c in range(len(floor[r])):
            if r == row and c == col:
                print(f" >{floor[r][c]}< ", end = '')
            else:
                print(f"  {floor[r][c]}  ", end = '')
        print(end = '\n')
    print(end = '\n')

# Test 1
floor = [[1, 0, 0, 0],
         [0, 1, 0, 1],
         [1, 0, 1, 1]]

print("Room Condition: ")
for row in floor:
    print(row)
print("\n")
clean(floor, 1, 2)
```

OUTPUT:

```
Room Condition:
[1, 0, 0, 0]
[0, 1, 0, 1]
[1, 0, 1, 1]


 1   0   0   0
 0   1  >0<  1
 1   0   1   1

 1   0   0   0
 0   1   0  >1<
 1   0   1   1

 1   0   0   0
 0   1   0  >0<
 1   0   1   1

 1   0   0   0
 0   1  >0<  0
 1   0   1   1

 1   0   0   0
 0  >1<  0   0
 1   0   1   1

 1   0   0   0
 0  >0<  0   0
 1   0   1   1

 1   0   0   0
 0   0   0   0
 1  >0<  1   1
```

```
 1   0   0   0
 0   0   0   0
>1<  0   1   1

 1   0   0   0
 0   0   0   0
>0<  0   1   1

 1   0   0   0
 0   0   0   0
 0  >0<  1   1

 1   0   0   0
 0   0   0   0
 0   0  >1<  1

 1   0   0   0
 0   0   0   0
 0   0  >0<  1

 1   0   0   0
 0   0   0   0
 0   0   0  >1<

 1   0   0   0
 0   0   0   0
 0   0   0  >0<

 1   0   0   0
 0   0   0  >0<
 0   0   0   0

 1   0   0  >0<
 0   0   0   0
 0   0   0   0
```

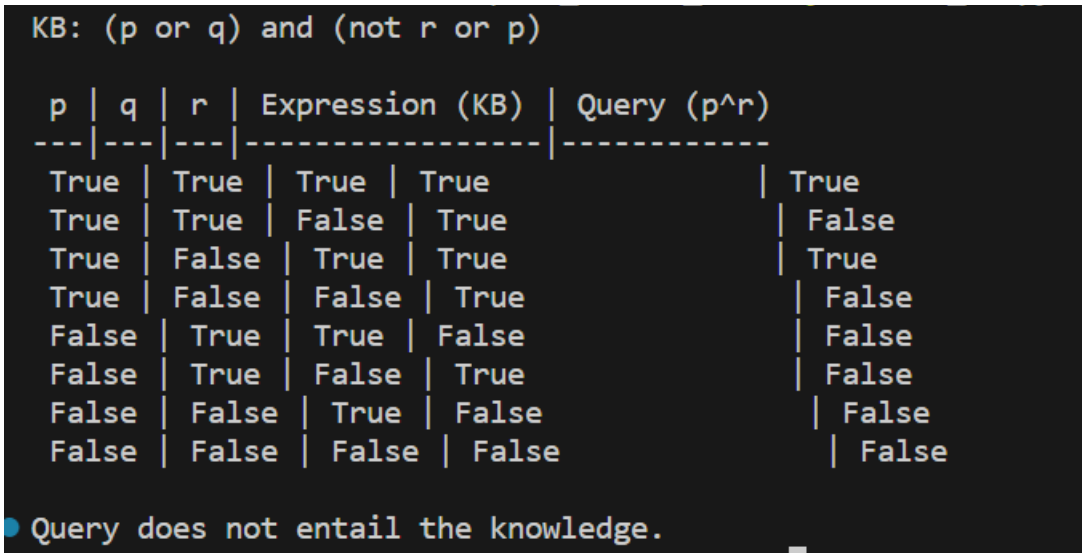
```
 1   0  >0<  0
 0   0   0   0
 0   0   0   0

 1  >0<  0   0
 0   0   0   0
 0   0   0   0

>1<  0   0   0
 0   0   0   0
 0   0   0   0

>0<  0   0   0
 0   0   0   0
 0   0   0   0
```

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

```
def evaluate_expression(p, q, r):
    expression_result = (p or q) and (not r or p)
    return expression_result

def generate_truth_table():
    print(" p | q | r | Expression (KB) | Query (p^r)")
    print("---|---|---|-----------------|------------")

    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r

                print(f" {p} | {q} | {r} | {expression_result}              | {query_result}")

def query_entails_knowledge():
    for p in [True, False]:
        for q in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(p, q, r)
                query_result = p and r

                if expression_result and not query_result:
                    return False

    return True
```

```
def main():
    generate_truth_table()

    if query_entails_knowledge():
        print("\nQuery entails the knowledge.")
    else:
        print("\nQuery does not entail the knowledge.")

if __name__ == "__main__":
    main()
```

OUTPUT:



7. **Create a knowledge base using prepositional 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')
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'
print('Rules: ',rules)
print("Goal: ",goal)
main(rules, goal)

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

rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
goal = 'R'
print('Rules: ',rules)
```

print("Goal: ",goal)

main(rules, goal)

OUTPUT:

```
Example 1
Rules:  Rv~P Rv~Q ~RvP ~RvQ
Goal:  R

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.

Example 2
Rules:  PvQ ~PvR ~QvR
Goal:  R

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.
```

```
Example 3
Rules:  PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
Goal:  R

Step    |Clause |Derivation
-----------------------------
 1.     | PvQ   | Given.
 2.     | PvR   | Given.
 3.     | ~PvR  | Given.
 4.     | RvS   | Given.
 5.     | Rv~Q  | Given.
 6.     | ~Sv~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PvQ and ~PvR.
 9.     | Pv~S  | Resolved from PvQ and ~Sv~Q.
 10.    | P     | Resolved from PvR and ~R.
 11.    | ~P    | Resolved from ~PvR and ~R.
 12.    | Rv~S  | Resolved from ~PvR and Pv~S.
 13.    | R     | Resolved from ~PvR and P.
 14.    | S     | Resolved from RvS and ~R.
 15.    | ~Q    | Resolved from Rv~Q and ~R.
 16.    | Q     | Resolved from ~R and QvR.
 17.    | ~S    | Resolved from ~R and Rv~S.
 18.    |       | Resolved ~R and R to ~RvR, 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

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression

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

print("\nExample 1")
exp1 = "knows(f(x),y)"
exp2 = "knows(J,John)"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)

print("\nExample 2")
exp1 = "knows(John,x)"
```

```
exp2 = "knows(y,mother(y))"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)


print("\nExample 3")
exp1 = "Student(x)"
exp2 = "Teacher(Rose)"
print("Expression 1: ",exp1)
print("Expression 2: ",exp2)

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

OUTPUT:

```
Example 1
Expression 1:  knows(f(x),y)
Expression 2:  knows(J,John)
Substitutions:
[('J', 'f(x)'), ('John', 'y')]

Example 2
Expression 1:  knows(John,x)
Expression 2:  knows(y,mother(y))
Substitutions:
[('John', 'y'), ('mother(y)', 'x')]

Example 3
Expression 1:  Student(x)
Expression 2:  Teacher(Rose)
Predicates do not match. Cannot be unified
Substitutions:
False
```

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~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)


def Skolemization(statement):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
    return statement
import re


def fol_to_cnf(fol):
    statement = fol.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
    return Skolemization(statement)

print(fol_to_cnf("bird(x)=>~fly(x)"))
print(fol_to_cnf("∃x[bird(x)=>~fly(x)]"))

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:

```
Example 1
FOL: bird(x)=>~fly(x)
CNF:  ~bird(x)|~fly(x)

Example 2
FOL: ∃x[bird(x)=>~fly(x)]
CNF:  [~bird(A)|~fly(A)]

Example 3
FOL: animal(y)<=>loves(x,y)
CNF:  ~animal(y)<|loves(x,y)

Example 4
FOL: ∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]
CNF:  ∀x~[∀y[~animal(y)|loves(x,y)]]|[[loves(A,x)]]

Example 5
FOL: [american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)
CNF:  ~[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()

kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

OUTPUT:

```
Example 1
Querying criminal(x):
        1. criminal(West)
All facts:
        1. american(West)
        2. enemy(Nono,America)
        3. hostile(Nono)
        4. sells(West,M1,Nono)
        5. owns(Nono,M1)
        6. missile(M1)
        7. weapon(M1)
        8. criminal(West)

Example 2
Querying evil(x):
        1. evil(John)
```