VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

VIGNESH (1BM21CS240)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled "**Artificial Intelligence**" carried out by **VIGNESH (1BM21CS240),** 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 June-2023 to Sep-2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

Dr. Kayarvizhy N

Associate Professor
Department of CSE
BMSCE, Bengaluru

Dr. Jyothi S Nayak

Professor and Head
Department of CSE
BMSCE, Bengaluru

Index Sheet

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.

1. **Implement Tic –Tac –Toe Game.**

Observation:

TIC TAC TOE

Minimax Algorithm - DFS

```
function findBestMove(board):
    bestMove = NULL;
    for each move in board:
        if current move is better than bestMove
            bestMove = current move

    return bestMove;


function minimax(board, depth, isMaxPlayer):

    if current board state is terminal state:
        return value of the board

    if isMaxPlayer:
        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
```

```
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.**

Observation:

Bafna Gold
Date: Page:

29/11/23

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 not in queue:
                queue.append(move)
```

```
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
```

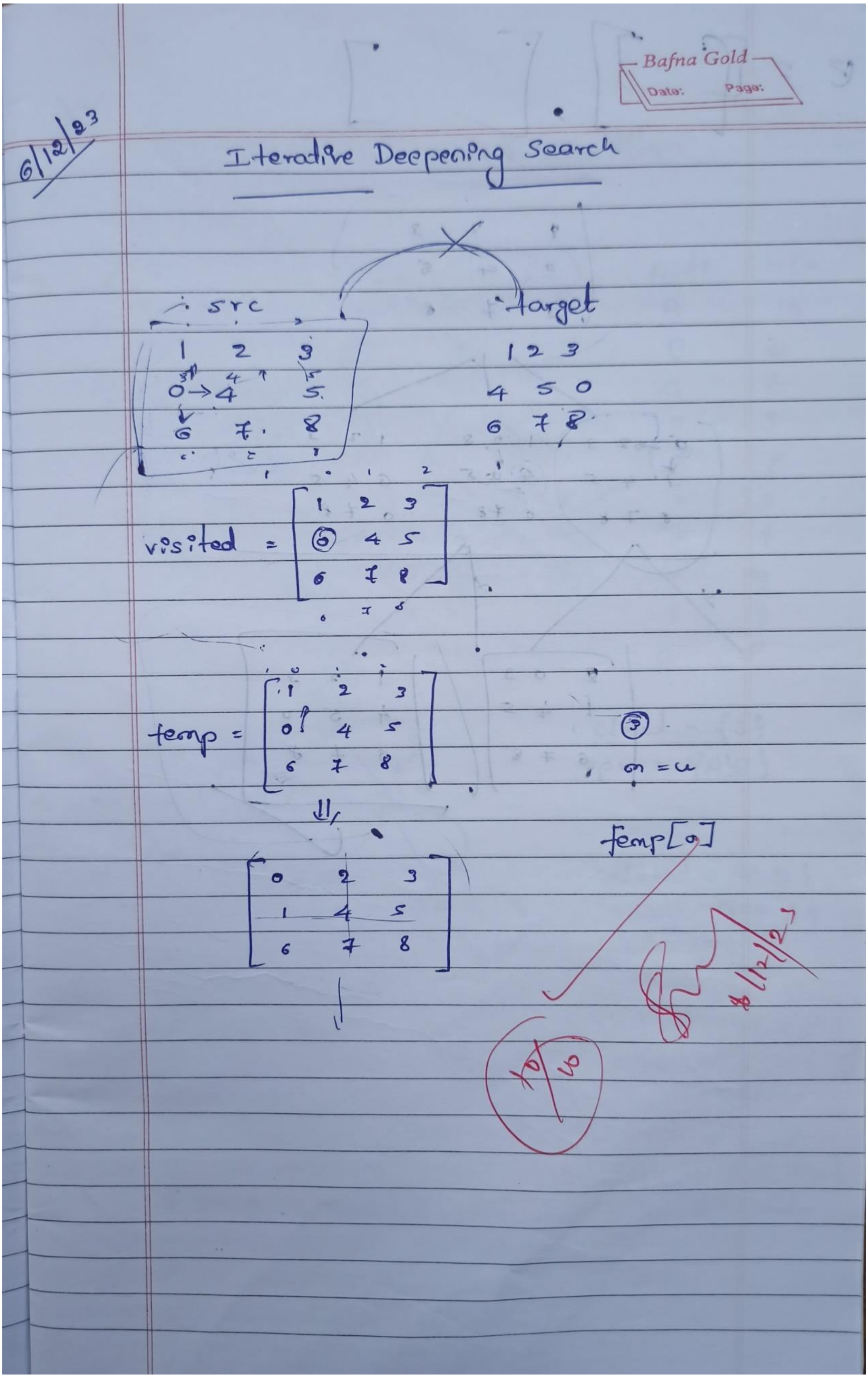
```
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
    if b not in [2,5,8]:
        d.append('r')


    # 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]
```

```python
    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
```

```python
    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)
```

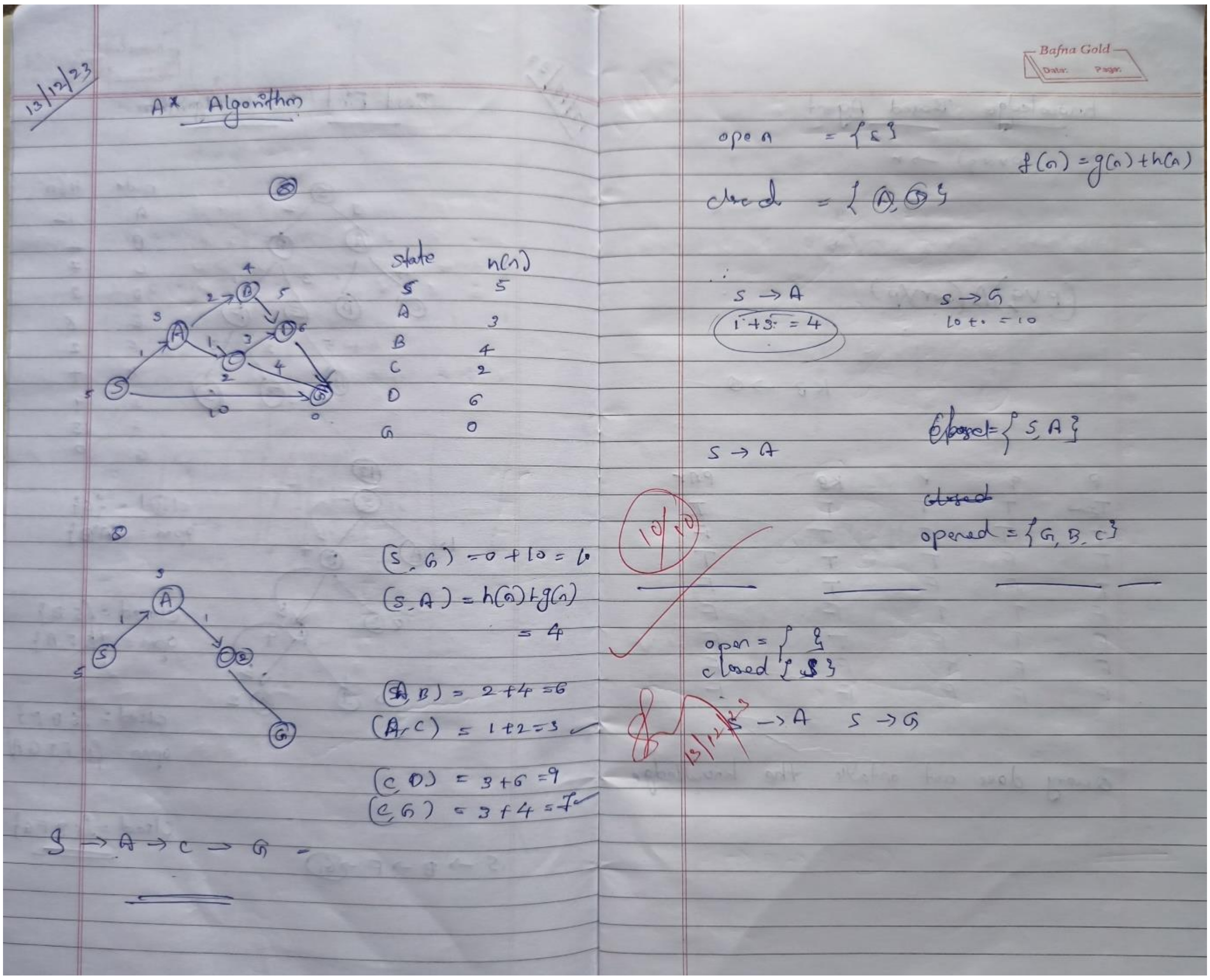
OUTPUT:

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

0 2 3
1 4 5
6 7 8

2 0 3
1 4 5
6 7 8

1 2 3
6 4 5
0 7 8

1 2 3
6 4 5
7 0 8

1 2 3
4 0 5
6 7 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

1 2 3
4 5 0
6 7 8

Success
```

4. Implement A* search algorithm.

Observation:



```
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.**

Observation:

Bafna Gold
Date: Page:

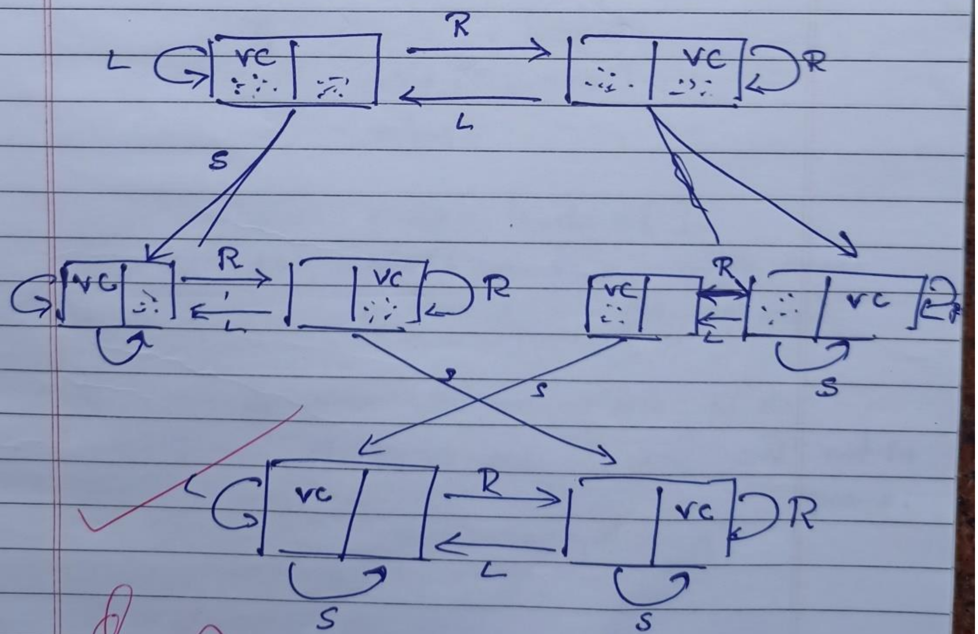
29/11/23

Implement Vacuum Cleaner Agent

Algorithm

function REFLEX-VACUUM-AGENT([Location, status]) returns action
    if status = Dirty then return Suck
    else if Location = A then return Right
    else if Location = B then return Left

State space Graph



```
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:

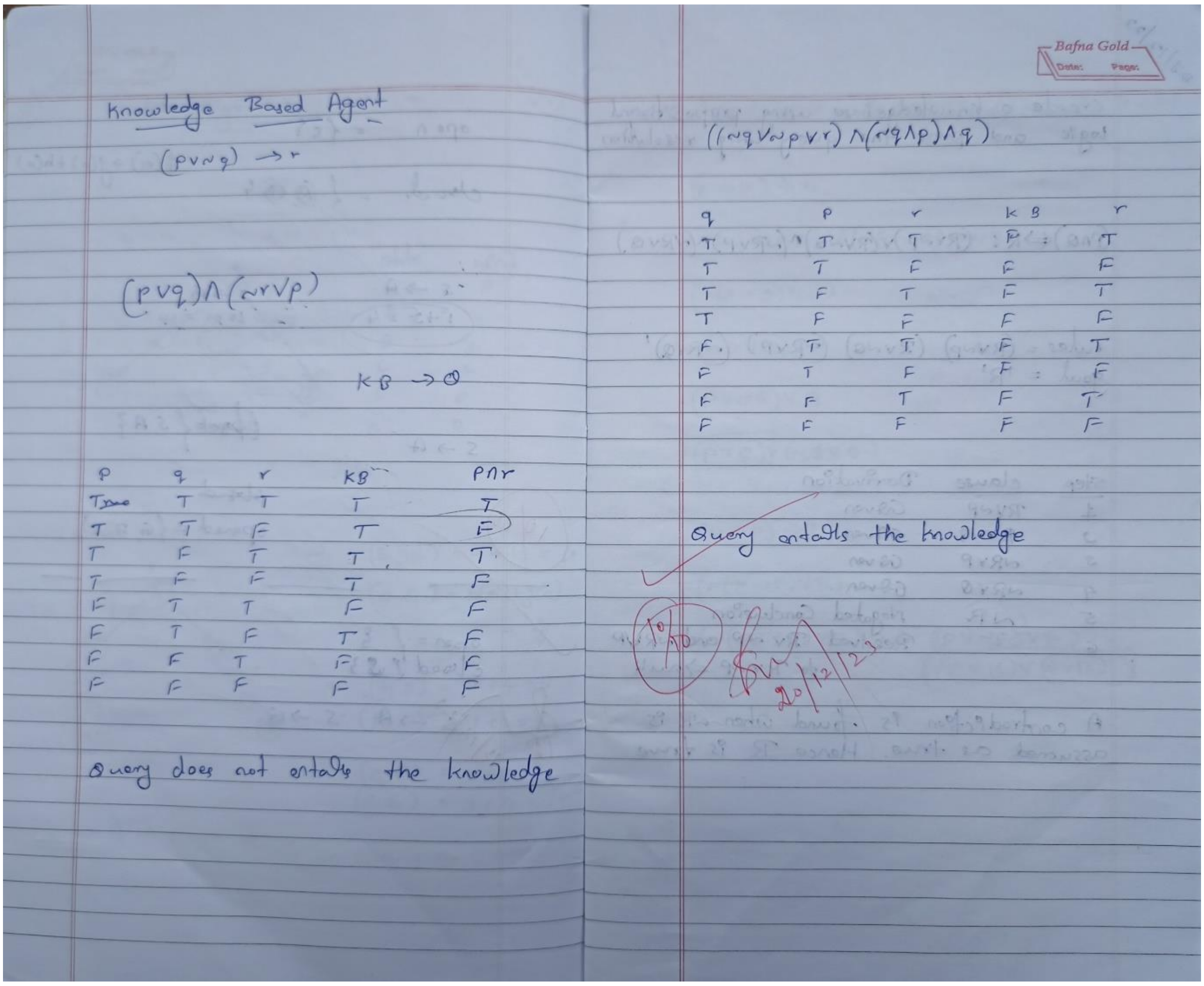
```
Room Condition:
[1, 0, 0, 0]
[0, 1, 0, 1]
[1, 0, 1, 1]


 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.**

Observation:




```
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]:
```

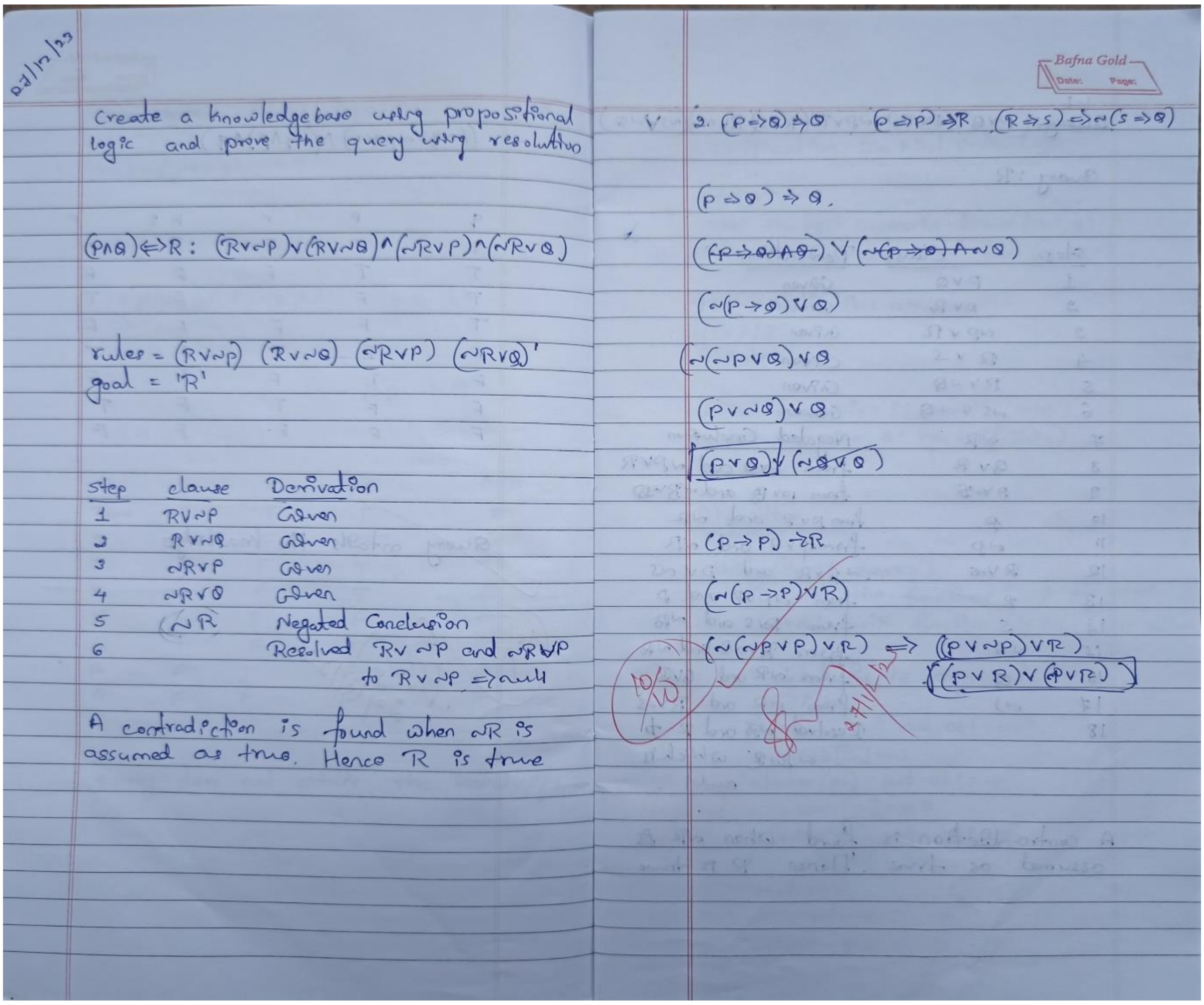
```
            expression_result = evaluate_expression(p, q, r)
            query_result = p and r

            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:

```
KB: (p or q) and (not r or p)

 p | q | r | Expression (KB) | Query (p^r)
---|---|---|-----------------|------------
 True | True | True | True               | True
 True | True | False | True               | False
 True | False | True | True               | True
 True | False | False | True               | False
 False | True | True | False               | False
 False | True | False | True               | False
 False | False | True | False               | False
 False | False | False | False               | False

Query does not entail the knowledge.
```

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

Observation:



27/11/23

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

(P∧Q)⇔R : (R∨~P)∨(R∨~Q)∧(~R∨P)∧(~R∨Q)

rules = (R∨~P) (R∨~Q) (~R∨P) (~R∨Q)'
goal = 'R'

Step	clause	Derivation
1	R∨~P	Given
2	R∨~Q	Given
3	~R∨P	Given
4	~R∨Q	Given
5	~R	Negated Conclusion
6		Resolved R∨~P and ~R∨P to R∨~P ⇒ null

A contradiction is found when ~R is assumed as true. Hence R is true

2. (P⇒Q)⇒Q   (P⇒P)⇒R   (R⇒S)⇒~(S⇒Q)

(P⇒Q)⇒Q.

(~(P→Q))∨Q

(~(~P∨Q))∨Q

(P∨~Q)∨Q

(P∨Q)∨(~Q∨Q)

(P→P)⇒R

(~(P→P)∨R)

(~(~P∨P)∨R) ⇒ ((P∨~P)∨R)

((P∨R)∨(~P∨R))

10/10

```
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

Observation:

Unification In First Order Logic

OUTPUT

Enter the first Expression: knows(f(x), y)

Enter the second Expression: knows(J, John)

The substitutions are:

f(x) / J

y / John

Unified expression 1: knows(J, John)

Unified expression 2: knows(J, John)

Algorithm

```
Unify(expr1, expr2)
    splits the functions and argument.
    check both function are same else return

    check for the arguments.
    ~~sub~~
  islower(a1, a2) and a1 != a2
      sub[a1] = a2
  islower(a1) and a1 != a2
      sub[a1] = a2
  islower(a2) and a1 != a2
      sub[a2] = a1
  a1 != a2
      return None.
  return sub
```

10

10/1/24

```
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).

Observation:

Bafna Gold
Date: Page:

First Order Logic to CNF Conversion

* Eliminate $\Leftrightarrow$, replacing
 $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$

* Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with
 $\sim\alpha \lor \beta$

Move $\neg$ inwards

* $\neg(\forall x p) \equiv \exists x \neg p$.
* $\neg(\exists x p) \equiv \forall x \neg p$
* $\neg(\alpha \lor \beta) \equiv \neg\alpha \land \neg\beta$
* $\neg(\alpha \land \beta) \equiv \neg\alpha \lor \neg\beta$
* $\neg(\neg\alpha) \equiv \alpha$

* Standardize variables apart by renaming them: each quantifier should use different variables

* Skolemize:
 each existential variable is replaced by a skolem constant or skolem function of enclosing universally quantified variables

· For instance $\exists x Rich(x)$ becomes $Rich(G1)$, where G1 is a new skolem constant.

```
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.

Observation:

~~sta~~

Forward Reasoning

FOL

missile(x) => weapon(x)
missile(M1)
enemy(x, America) => hostile(x)
american(west)
enemy(Nono, America)
owns(Nono, M1)
missile(x) & owns(Nono x) => sells(west, x, Nono)
american(x) & weapon(y) & sells(x, y, z) &
hostile(z) => criminal(x)

Query: criminal(x)

OUTPUT

10

Querying Criminal(x)

1. Criminal(west)

24/11/24

All facts:

1. hostile(Nono)
2. owns(Nono, M1)
3. enemy(Nono, America)
4. missile(M1)
5. Sells(West, M1, Nono)

```
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)
```