VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

KAMESH CHANDRA (1BM21CS271)

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 **KAMESH CHANDRA (1BM21CS271),** 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 a **Artificial Intelligence (22CS5PCAIN)** work prescribed for the said degree.

M Lakshmi Neelima

Assistant 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.

```
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]:
```

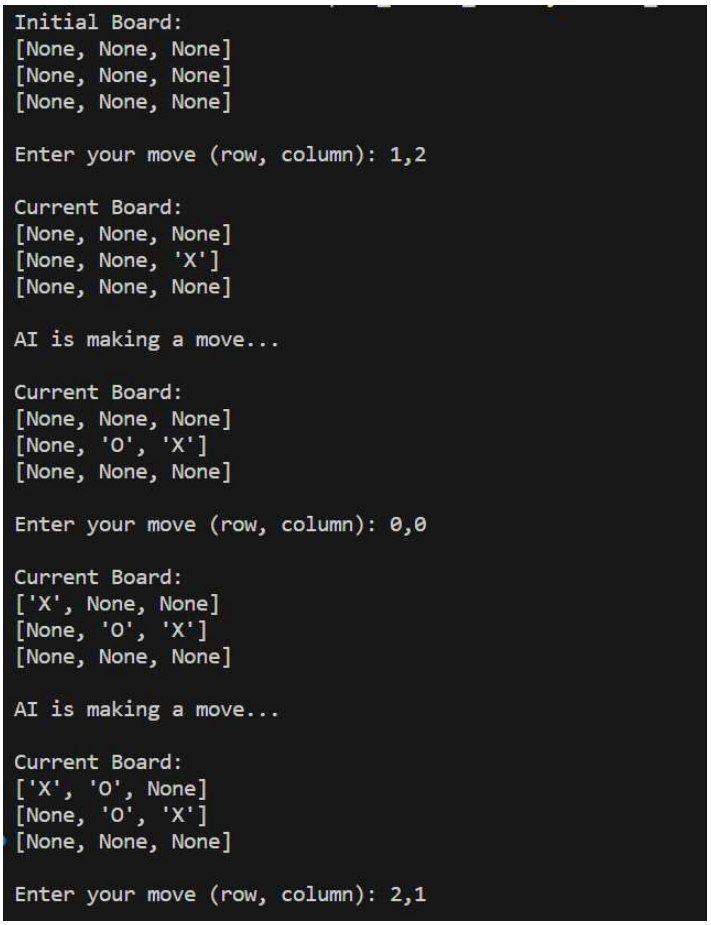
```
            s2.append(board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]
    strikeD = []
    for i in [0, 1, 2]:
        strikeD.append(board[i][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        return strikeD[0]
    if (board[0][2] == board[1][1] == board[2][0]):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

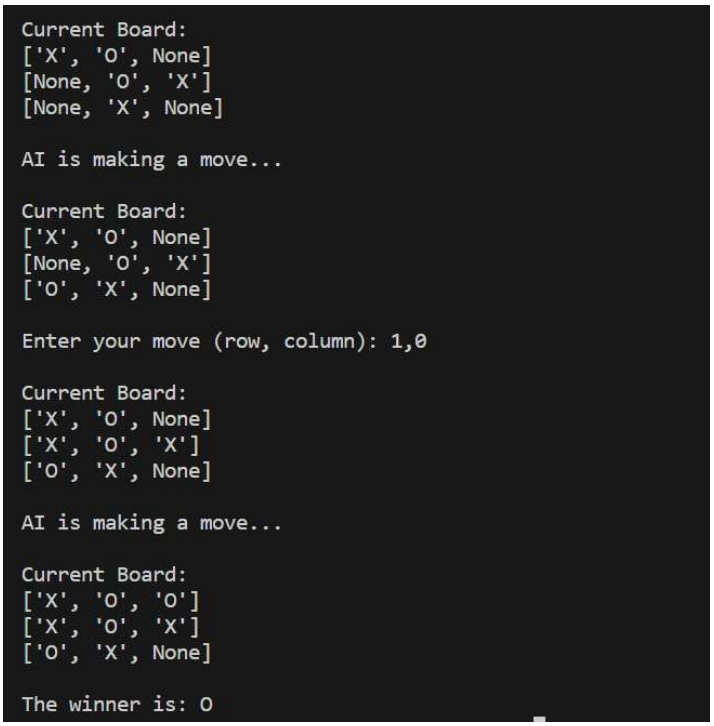
```
        return -1
    else:
        return 0


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:





Program 3

3. Explore the working of Tic Tac Toe using min max strategy

code

```
def constBoard(board):
    print("current state of Board:\n\n")
    for i in range(0,9):
        if((i>0) and (i%3)==0):
            print("\n")
        if(board[i]==0):
            print("-",end=" ")
        if(board[i]==1):
            print("0",end=" ")
        if(board[i]==-1):
            print("X",end=" ")
    print("\n\n")
def userTurn(board):
    pos=input("Enter X's position from[1...9]:")
    Pos=int(pos)
    if(board[Pos-1]!=0):
        print("wrong move!!")
        exit(0)
    Board[Pos-1]=-1
def user2Turn(board):
    pos=input("Enter o's Position from[1...9]:")
    Pos=int(Pos)
    if(board[Pos-1]!=0):
        print("wrong move!!")
        exit(0)
    board[Pos-1]=1;
```

```
def minimax(board, player):
    x = analyzeboard(board);
    if (x != 0)
        return (x * player);
    pos = -1;
    value = -2;
    for i in range(0, 9):
        if (board[i] == 0):
            board[i] = player;
            score = -minimax(board, (player * -1));
            if (score > value):
                value = score;
                pos = i;
            board[i] = 0;

    if (pos == -1)
        return 0;
    return value;
def compTurn(board):
    pos = -1;
    value = -2;
    for i in range(0, 9):
        if (board[i] == 0):
            board[i] = 1;
            score = -minimax(board, -1);
            board[i] = 0;
            if (score > value):
                value = score;
                pos = i;
    board[pos] = 1;
```

```
def analyzeboard(board):
    cb = [[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],
          [2,5,8],[0,4,8],[2,4,6]]

    for i in range(0, 8):
        if (board[cb[i][0]] != 0 and
            board[cb[i][0]] == board[cb[i][1]] and
            board[cb[i][0]] == board[cb[i][2]]):
                return board[cb[i][2]];
    return 0;
def main():
    choice = input("Enter 1 for single Player
                    2 for multiplayer: ");
    choice = int(choice);
    board = [0,0,0,0,0,0,0,0,0];
    if (choice == 1):
        print("computer: O VS You: X");
        player = input("Enter To play 1st or 2nd: ");
        player = int(player)
        for i in range(0, 9):
            if (analyzeboard(board) != 0):
                break;
            if ((i + player) % 2 == 0):
                compTurn(board);
            else:
                printBoard(board)
                userTurn(board);
    else:
        for i in range(0,9):
            if (analyzeboard(board) != 0):
                break.
            if ((i) % 2 == 0):
                printBoard(board);
                userTurn(board);
            else:
```

```
        constBoard(board);
        userTurn(board);

    x = analyzeboard(board);
    if (x==0):
        constBoard(board);
        print("Draw!!!")
    if (x==-1):
        constBoard(board);
        print("X wins!!! & loose!!!")
    if (x==1):
        constBoard(board);
        print("X loose!!! O wins!!!")
#.....#
main()
#.....#
```

Output :

```
Enter 1 for single player, 2 for multiplayer:1
computer : O vs You : X
Enter to play 1(st) or 2(nd) : 1
Current state of Board:
_ _ _
_ _ _
_ _ _

Enter X's position from [1.....9] : 2
Current State of Board
O X _
_ _ _
_ _ _
```

Enter x's position from [1...9]: 3
Current state of Board:
O X X
O - -
- - -

Enter X's position from [1...9]: 6
Current state of Board:
O X X
O - X
O - -

X loose!!! O wins!!!

NL
18/12/2023

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
```

Program 2

2. Implement the 8 Puzzle Breadth First Search Algorithm.

```
import sys
import numpy as np
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
class StackFrontier:
    def __init__(self):
        self.frontier = []
    def add(self, node):
        self.frontier.append(node)
    def contains_state(self, state):
        return any((node.state[0] == state[0]).all() for
                   node in self.frontier)
    def empty(self):
        return len(self.frontier) == 0
    def remove(self):
        if self.empty():
            raise Exception("EmptyFrontier")
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty()
            raise Exception("Empty Frontier")
        else:
```

```
        node = self.frontier[-1]
        self.frontier = self.frontier[:-1]
        return node
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None
    def neighbors(self, state):
        mat, (row, col) = state
        results = []
        if row > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row-1][col]
            mat1[row-1][col] = 0
            results.append(('up', [mat1, (row-1, col)]))
        if col > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row][col-1]
            mat1[row][col-1] = 0
            results.append(('left', [mat1, (row, col-1)]))
        if row < 2:
            mat1 = np.copy(mat)
```

```
        mat1[row][col] = mat1[row+1][col]
        mat1[row+1][col] = 0
        results.append(('down', [mat1, (row+1, col)]))
    if col < 2:
        mat1 = np.copy(mat)
        mat1[row][col] = mat1[row][col+1]
        mat1[row][col+1] = 0
        results.append(('right', [mat1, (row, col+1)]))
    return results

def print(self):
    solution = self.solution if self.solution is not None else None
    print("Start State:\n", self.start[0], "\n")
    print("Goal State:\n", self.goal[0], "\n")
    print("\nStates Explored: ", self.num_explored)
    print("Solution:\n")
    for action, cell in zip(solution[0], solution[1]):
        print("action", action, "\n", cell[0], "\n")
    print("Goal Reached")
else:
    node = self.frontier[-1]
    self.frontier = self.frontier[:-1]
    return node
class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```

```
class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None
    def neighbours(self, state):
        mat, (row, col) = state
        results = []
        if row > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row-1][col]
            mat1[row-1][col] = 0
            results.append(('up', [mat1, (row-1, col)]))
        actions = reverse()
        cells = reverse()
        self.solution = (actions, cells)
        return

    self.explored.append(node.state)

    for action, state in self.neighbours(node.state):
        if not frontier.contains(state) and
            self.does_not_contain_state(state):
            child = Node(state=state, parent=node,
                         action=action)
            frontier.add(child)

start = np.array([[1,2,3],[8,0,4],[7,6,5]])
goal = np.array([[2,8,1],[0,4,3],[7,6,5]])
startIndex = (1,1)
goalIndex = (1,0)
```

Program 2 - 02

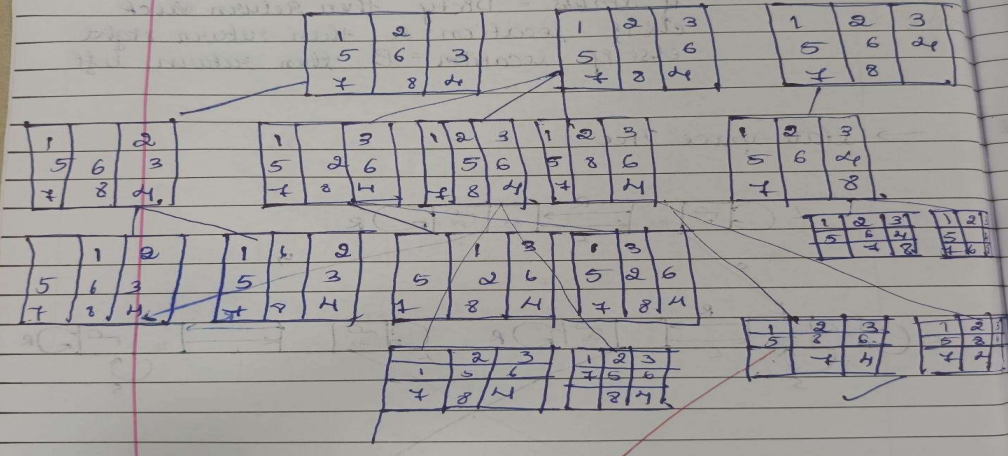
Initial state

1	2	3
5	6	
7	8	4

Final State

1	2	3
5	8	6
	7	4

Breadth first



Best - First

f(n) = h(n)

h(n): no of misplaced tiles.

p= Puzzle(start, startIndex, goal, goalIndex)
p.solve()
p.print()

output:

start state:

[[1 2 3]
 [8 0 4]
 [7 6 5]]

goal state:
[[2 8 1]
 [0 4 3]
 [7 6 5]]

solution

action: up
[[1 0 3]
 [8 2 4]
 [7 6 5]]
action: left
[[0 1 3]
 [8 2 4]
 [7 6 5]]
action: down
[[8 1 3]
 [0 2 4]
 [7 6 5]]
action: right

[[8 1 3]
 [2 0 4]
 [7 6 5]]
action: right
[[8 1 3]
 [2 4 0]
 [7 6 5]]
action: up
[[8 1 0]
 [2 4 3]
 [7 6 5]]
action: left
[[8 0 1]
 [2 4 3]
 [7 6 5]]
action: left
[[0 8 1]
 [2 4 3]
 [7 6 5]]
action: down
[[2 8 1]
 [0 4 3]
 [7 6 5]]

18/12/2022

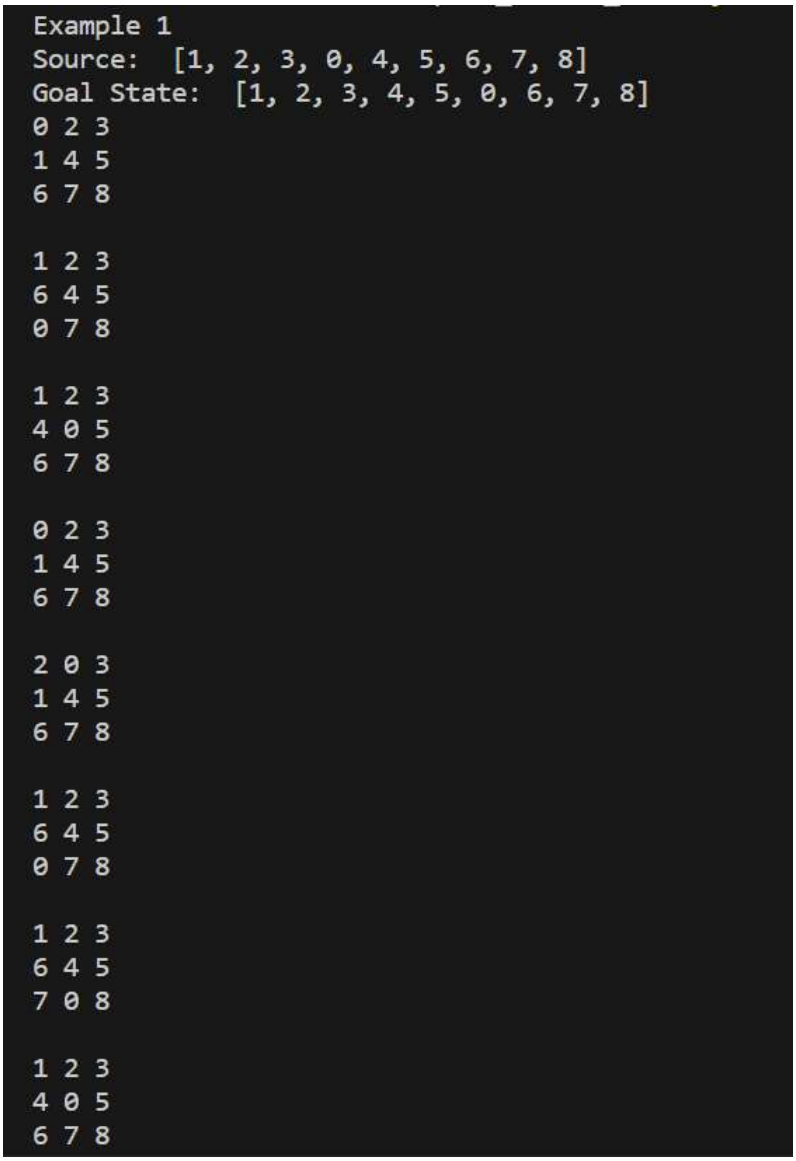
3. Implement Iterative deepening search algorithm.

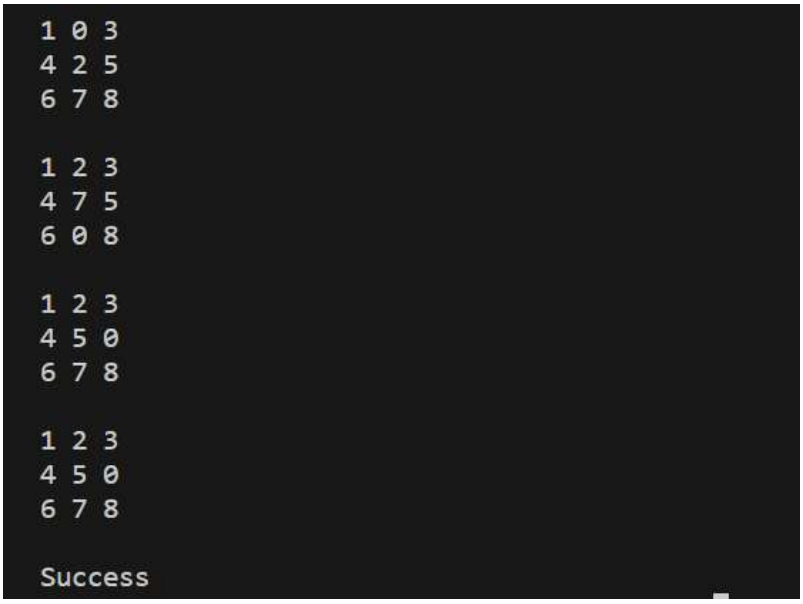
```
def iterative_deepening_search(src, target):
    depth_limit = 0
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is not None:
            print("Success")
            return
        depth_limit += 1
        if depth_limit > 30: # Set a reasonable 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:


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

Program 4

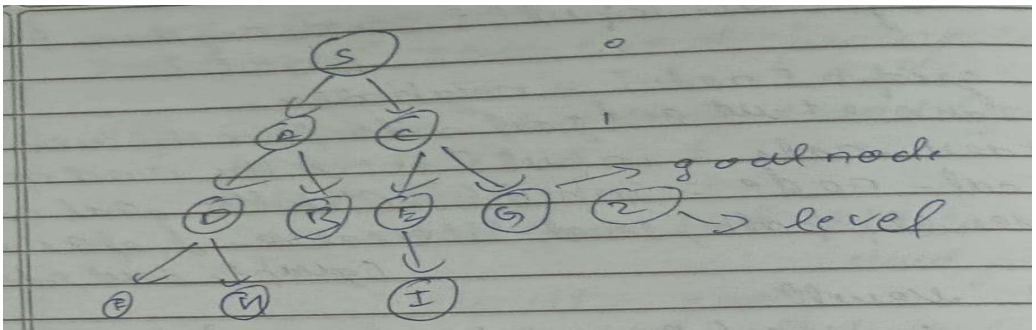
9. Implement Iterative deepening search algorithm

→
```
def iterative_deepening_search(graph, start, goal):
    depth_limit = 0
    while True:
        result, path, level = depth_limited_search(graph,
                                start, goal, depth_limit, [start], 0)
        if result == goal:
            return result, path, level
        depth_limit += 1
def depth_limited_search(graph, current, goal, depth_limit, path,
                         level):
    if current == goal:
        return current, path, level
    if depth_limit == 0:
        return None, [], 0
    if depth_limit > 0:
        for neighbour in graph[current]:
            result, new_path, new_level = depth_limited
                search(graph, neighbour, goal,
                    depth_limit - 1, path+[neighbour],
                    level + 1)
            if result == goal:
                return result, new_path, new_level
    return None, [], 0
def main():
    # Take graph input from the user
    graph = {}
    while True:
        node = input("Enter a node (or 'done' to finish): ")
        if node.lower() == 'done':
            break
```

```
        neighbors = input(f"Enter neighbors for {node}: ").split()
        graph[node] = neighbors
# Take start and goal nodes from the user
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
result, path, level = iterative_deepening_search(graph, start_node, goal_node)
if result:
    print(f"Goal '{goal_node}' found at level {level}")
    print(f"Path: {path}")
else:
    print(f"Goal '{goal_node}' not found.")
if __name__ == "__main__":
    main()
```

Output

```
Enter a node (or 'done' to finish): S
Enter neighbors for S: A C
Enter a node (or 'done' to finish): A
Enter neighbors for A: D B
Enter a node (or 'done' to finish): C
Enter neighbors for C: E G
Enter a node (or 'done' to finish): D
Enter neighbors for D: F H
Enter a node (or 'done' to finish): E
Enter neighbors for E: I
Enter a node (or 'done' to finish): done
Enter the start node: S
Enter the goal node: G
Goal 'G' found at level 2. Path: ['S', 'C', 'G']
```



18/12/2023

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:
```

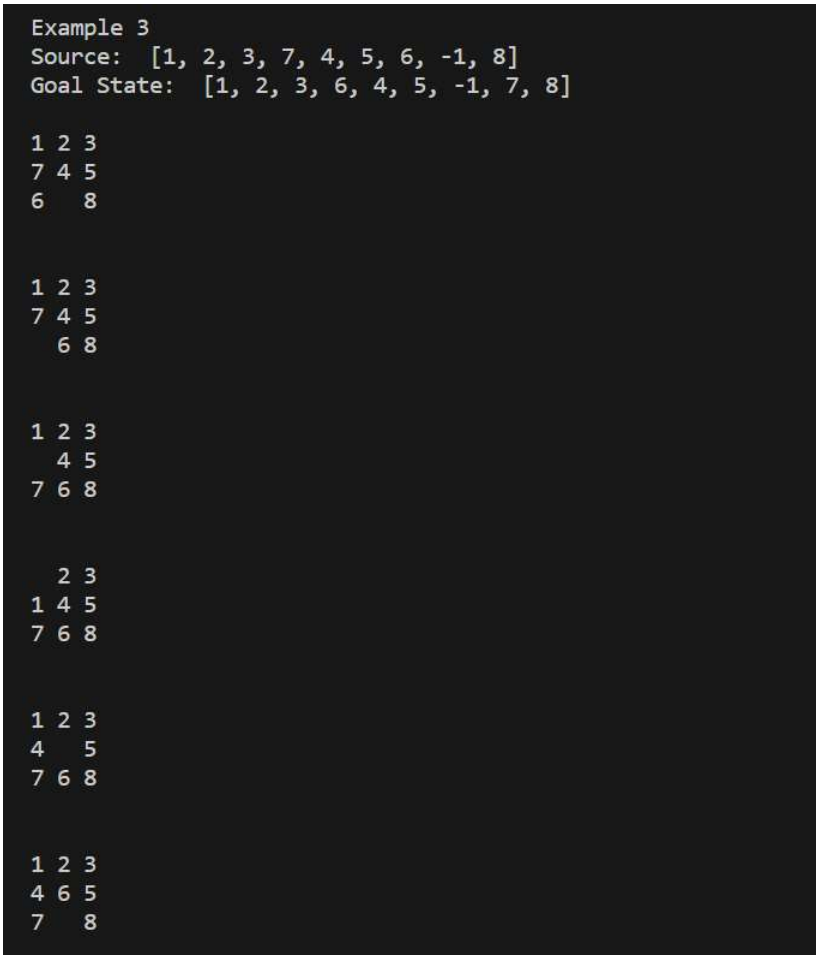
```
            visited_states.add(tuple(state))
            print_grid(state)
            if state == target:
                print("Success")
                return
            moves += [move for move in possible_moves(state, visited_states) if move not in
moves]

        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':
```

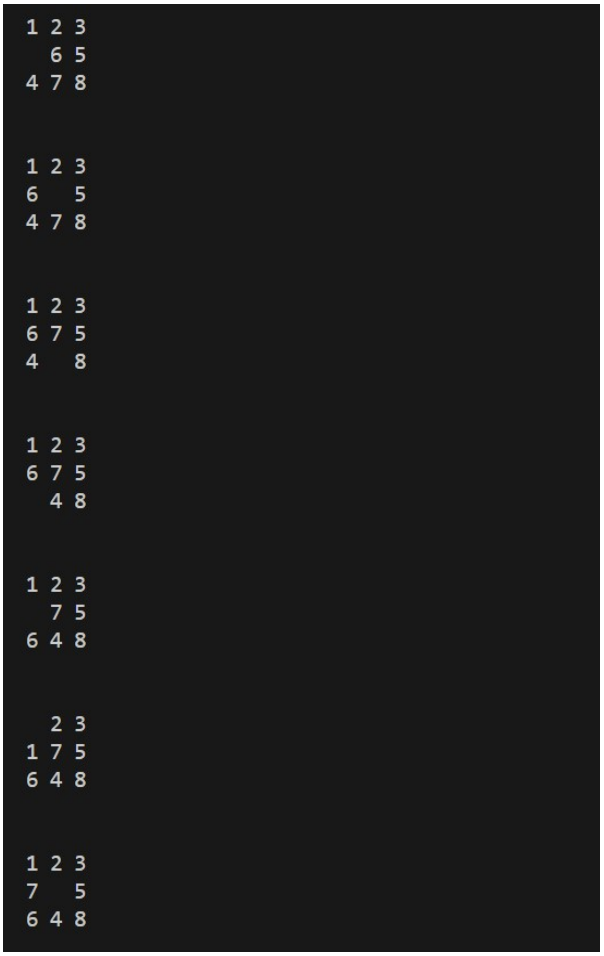
```
        temp[b-3], temp[b] = temp[b], temp[b-3]
    if direction == 'd':
        temp[b+3], temp[b] = temp[b], temp[b+3]
    if direction == 'r':
        temp[b+1], temp[b] = temp[b], temp[b+1]
    if direction == 'l':
        temp[b-1], temp[b] = temp[b], temp[b-1]
    return temp

#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]
```

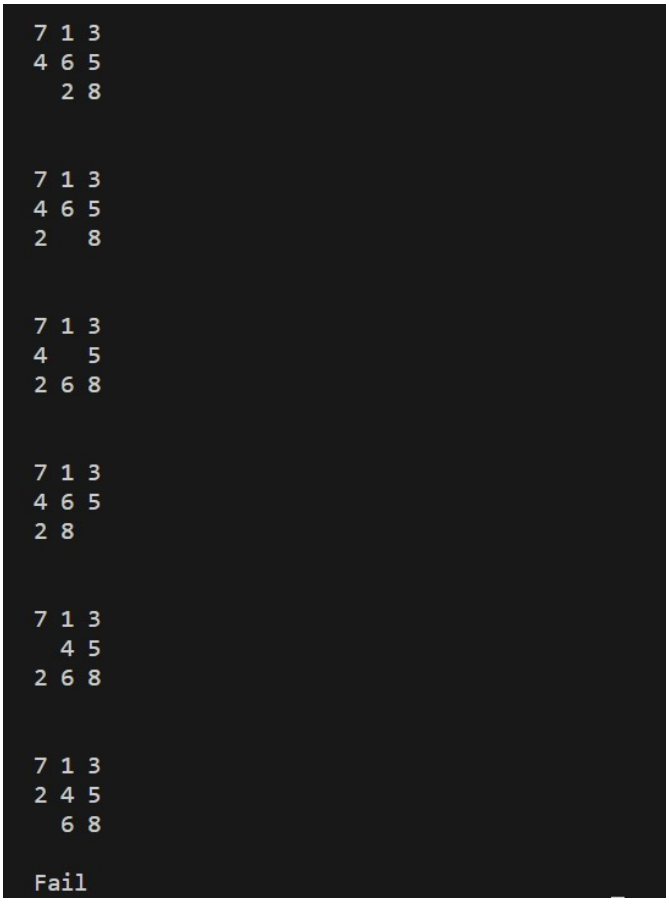
```
target=[1,2,3,6,4,5,-1,7,8]
print("Source: " , src)
print("Goal State: " , target)
astar(src, target)
```

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

Program 5

5. Implement A* for 8 puzzle Problem

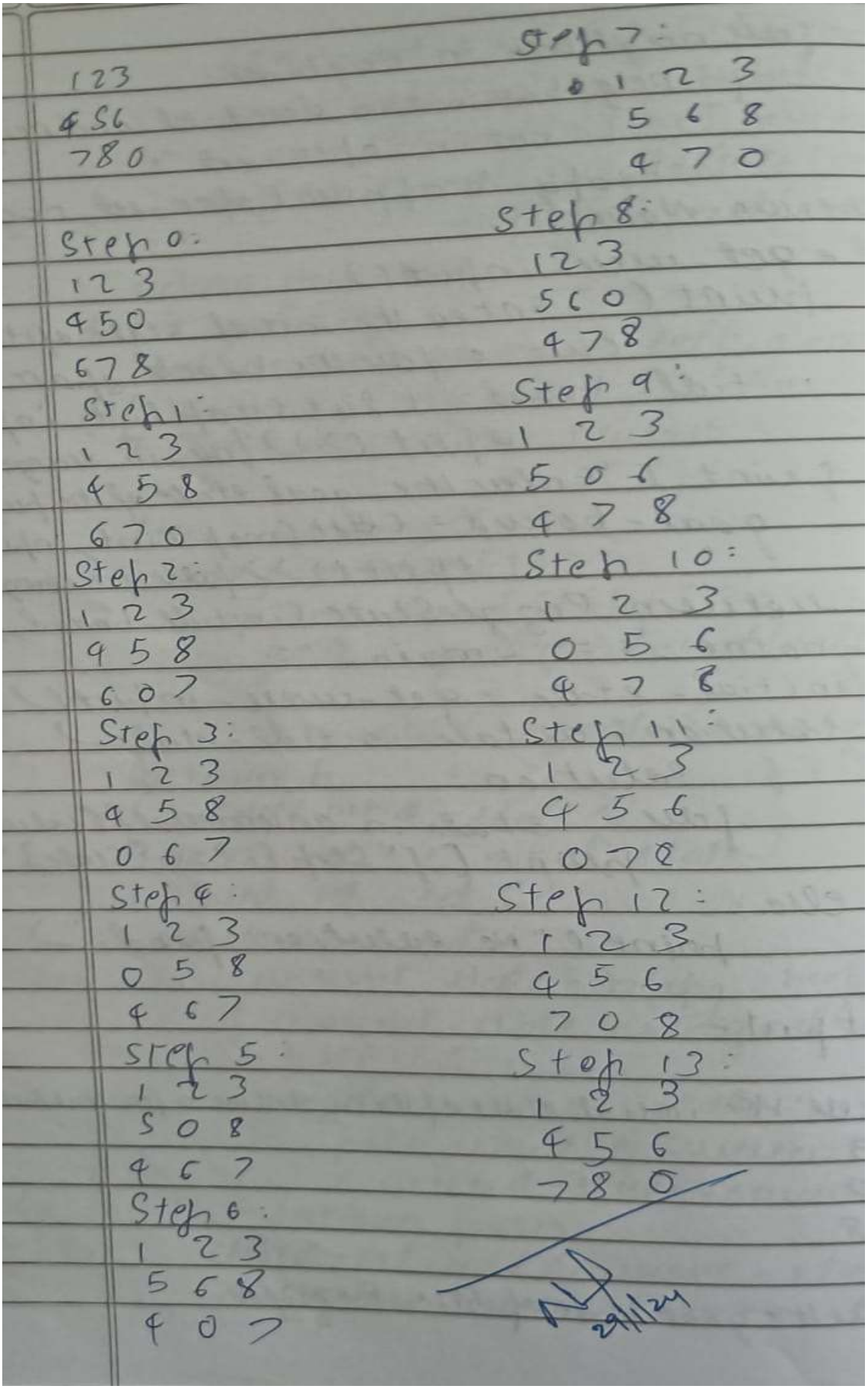
```
import heapq
class PuzzleState:
    def __init__(self, board, goal, cost=0, parent=None):
        self.board = board
        self.goal = goal
        self.cost = cost
        self.parent = parent
    def __lt__(self, other):
        return (self.cost+self.heuristic()) < (other.cost
                                    + other.heuristic())

    def __eq__(self, other):
        return self.board == other.board
    def __hash__(self):
        return hash(tuple(map(tuple, self.board)))
    def __str__(self):
        return '\n'.join([' '.join(map(str, row)) for
                                    row in self.board])
    def get_blank_position(self):
        for i, row in enumerate(self.board):
            for j, tile in enumerate(row):
                if tile == 0:
                    return i, j
    def get_neighbors(self):
        i, j = self.get_blank_position()
        neighbors = []
        for move in [(0,1), (1,0), (0,-1), (-1,0)]:
            ni, nj = i+move[0], j+move[1]
            if 0<=ni<3 and 0<=nj<3:
```

```
        new_board = [row.copy() for row in self.board]
        new_board[i][j], new_board[ni][nj] =
                new_board[ni][nj], new_board[i][j]
        neighbors.append(PuzzleState(new_board,
                self.goal, self.cost+1, self))
    return neighbors
def is_goal(self):
    return self.board == self.goal
def heuristic(self):
    # Manhattan distance heuristic
    h = 0
    for i in range(3):
        for j in range(3):
            if self.board[i][j] != 0:
                goal_i, goal_j = divmod(self.board[i][j]-1,
                                        3)
                h += abs(i - goal_i) + abs(j - goal_j)
    return h
def astar(initial_state):
    open_set = [initial_state]
    closed_set = set()
    while open_set:
        current_state = heapq.heappop(open_set)
        if current_state.is_goal():
            path = []
            while current_state:
                path.insert(0, current_state)
                current_state = current_state.parent
            return path
        closed_set.add(current_state)
        neighbors = current_state.get_neighbors()
```

```
        for neighbor in neighbors:
            if neighbor not in closed_set and neighbor
                    not in open_set:
                heapq.heappush(open_set, neighbor)
    return None
def get_user_input():
    print("Enter the initial state of the puzzle
            (use 0 for the blank space):")
    initial_board = [list(map(int, input().
            split())) for _ in range(3)]
    print("Enter the goal state of the puzzle:")
    goal_board = [list(map(int, input().
            split())) for _ in range(3)]
    return PuzzleState(initial_board, goal_board)
if __name__ == "__main__":
    initial_state = get_user_input()
    solution = astar(initial_state)
    if solution:
        for i, state in enumerate(solution):
            print(f"Step {i}:\n{state}\n")
    else:
        print("No solution found.")
```

Output :

```
Enter the initial state of the Puzzle (use 0 for the blank space):
1 2 3
4 5 0
6 7 8

Enter the goal state of the Puzzle:
```


123
456
780
Step 0:
123
450
678
Step 1:
123
458
670
Step 2:
123
458
607
Step 3:
123
458
067
Step 4:
123
058
467
Step 5:
123
508
467
Step 6:
123
568
407
Step 7:
0123
568
470
Step 8:
123
560
478
Step 9:
123
506
478
Step 10:
123
056
478
Step 11:
123
456
078
Step 12:
123
456
708
Step 13:
123
456
780
29/11/24

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
```

Program:

1. Implement vacuum cleaner for 2 rooms using any type of agent

```
def vaccuum_world():
    goal_state = {'A':'0', 'B':'0'}
    cost = 0
    location_input = input("Enter location of vaccuum")
    status_input = input("Enter status of" + location_input);
    status_input_complement = input("Enter status of the room")

    if location_input == 'A':
        print("Vacuum is placed in location A")
        if status_input == '1':
            print("Location A is Dirty")
            goal_state['A'] = 'D'
            cost += 1
            print("Cost for cleaning A" + str(cost))
            print("Location A has been cleaned.")
        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 is already clean")
        else:
            print("No action" + str(cost))
            print("location B is already clean")
    if status_input == 0:
```

```
print ("Vacuum is placed in location B")
if status - input == '1':
    print ("location B is dirty")
    goal - state['B'] = '0'
    cost += 1
    print ("cost for cleaning" + str(cost))
    print ("location B has been changed")
else:
    print ("No action" + str(cost))
    print (cost)
    print ("location B is already clean")
else:
    print ("Vacuum is placed in location B")
    if status - input == '1':
        print ("location B is Dirty")
        goal - state['B'] = '0'
        cost += 1
        print ("cost 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))
        goal - state['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 is already 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()                    # calling function
```

Output

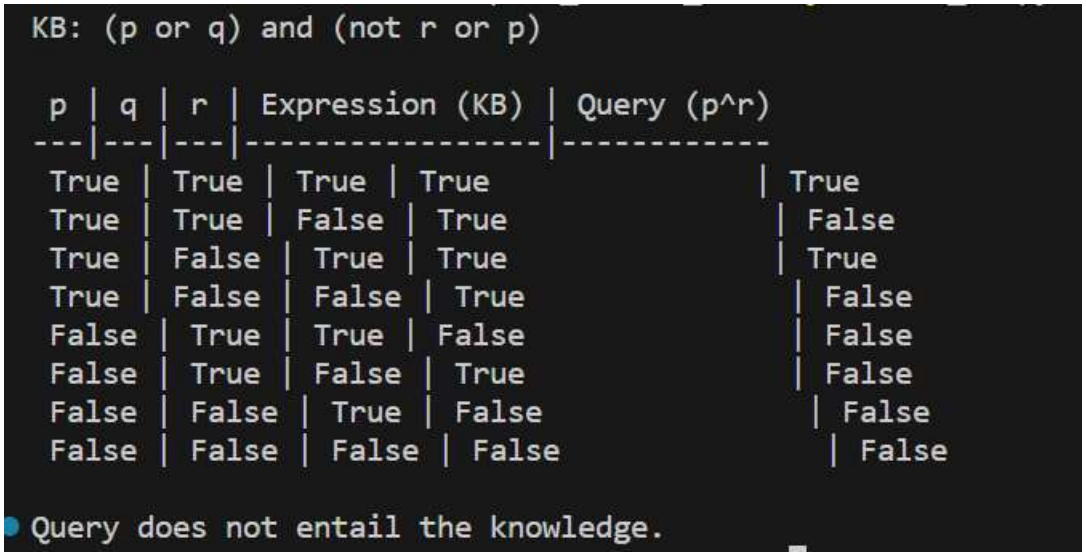
Enter location of vacuum A
Enter status of A 1
Enter status of other room 0
vacuum is placed in location A
Location A is dirty
cost for cleaning A 1
Location A has been cleaned
Location B is Dirty
moving right to the location B
cost for moving Right 2
cost for Suck 5
Location B has been cleaned
Vacuum is placed in location B

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:



Program - 6

Create a knowledge base using propositional logic & show that the give query entails the knowledge base or not.

Algorithm:

function TT-Entails ? (KB, q) returns true or false
inputs : KB, the knowledge base
q, the query as sentence
symbols : a list of the propositional symbol in KB & q

function TT-CHECK-ALL (KB, q, symbol, model)
return true or false
if Empty ? (symbol) then
if PL-True ? (KB, model) then return PL-True ? (a, model)
else return true.
else do
P = First (symbols); rest = Rest (symbols)
return TT-CHECK-ALL (KB, q, rest, EXTEND (P, true, model) and
TT-CHECK-ALL (KB, q, rest, EXTEND (P, false, model))

KB &
KB : $(p \wedge q) \vee (\neg p \wedge r)$
query : $p \vee q$

Program 6

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

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

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

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

    return knowledge_base

def query_entails(knowledge_base, query):
    entailment = satisfiable(And(knowledge_base, Not(query)))

    return not entailment

if __name__ == "__main__":
    kb = create_knowledge_base()
```

```
query = symbol('P')

result = query_entails(kb, query)

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

Output:

```
Knowledge Base: ~(P & Q) & (Implies(P, Q)) & (Implies(Q, P))
Query: P
Query entails Knowledge Base: False
```

29/11/24

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.
```

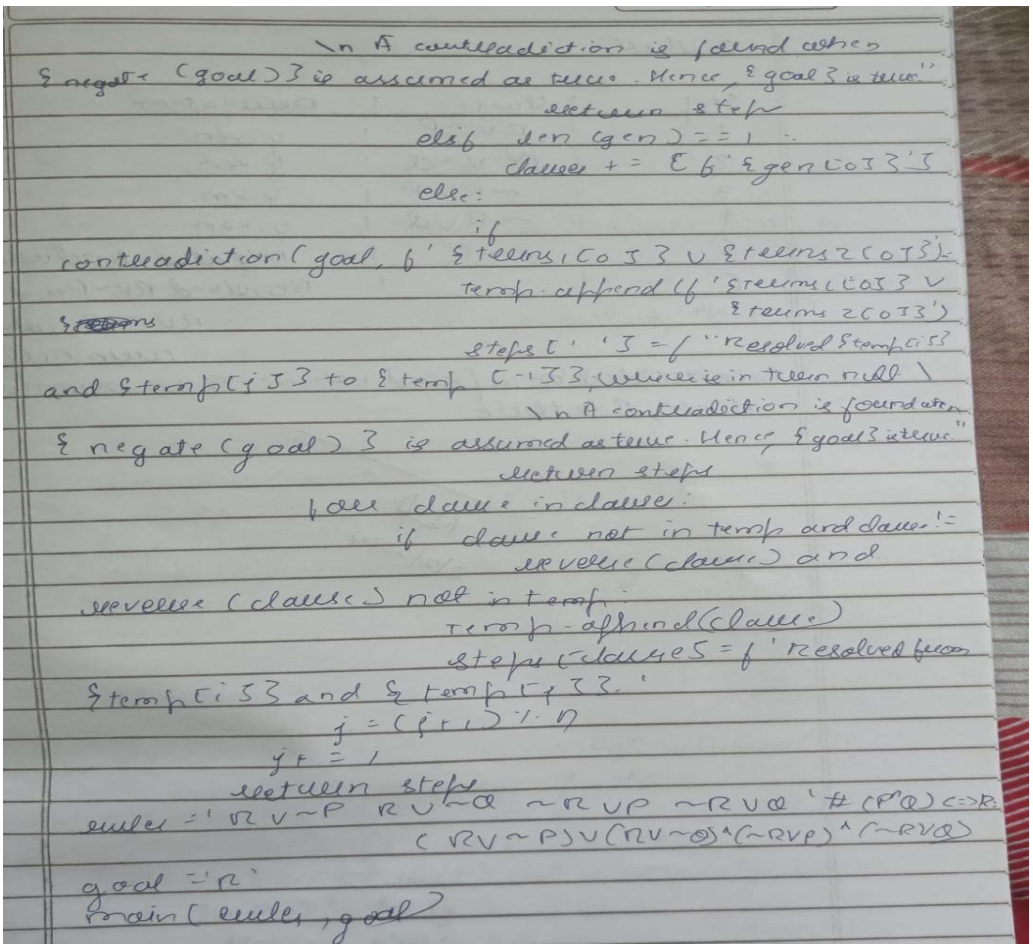
```
Example 3
Rules:  PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
Goal:  R

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.
```

Program-7

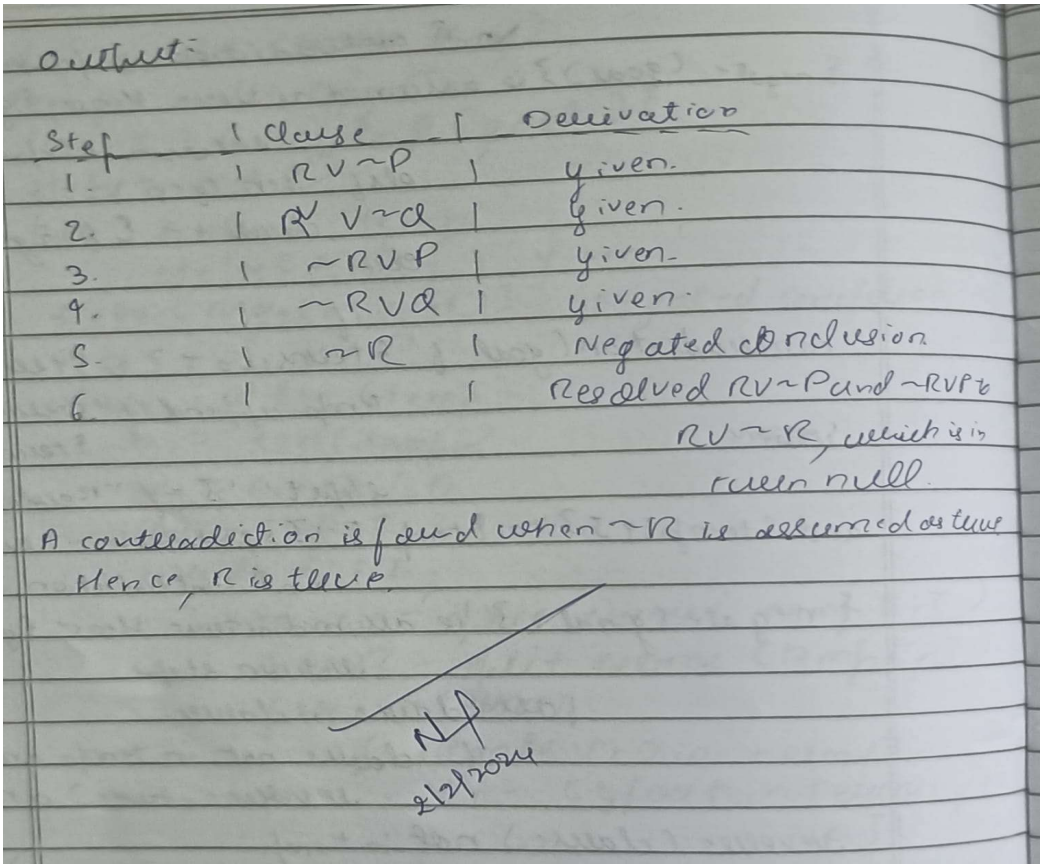
7. Create a knowledge base using propositional logic and prove the given query using resolution.

```
import re
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f'{i}.\t| {step}\t| {steps[step]}\t')
        i = 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('~P v R')
def contradiction(goal, clause):
    contradictions = [f'{goal} v {negate(goal)}', f'{negate(goal)} v {goal}']
    return clause in contradictions or reverse(clause) in contradictions
```

```
def resolve(rules, goal):
temp = rules.copy()
temp+ = [negate(goal)]
steps = dict()
for rule in temp:
    steps[rule] = 'given.'
steps[negate(goal)] = 'Negated conclusion.
i=0
while i < len(temp):
    n = len(temp)
    j = (i+1) % n
    clauses = []
    while j != i:
        terms1 = split_terms(temp[i])
        terms2 = split_terms(temp[j])
        for c in terms1:
            if negate(c) in terms2:
                t1 = [t for t in terms1 if t != c]
                t2 = [t for t in terms2 if t != negate(c)]
                gen = t1 + t2
                if len(gen) == 2:
                    if gen[0] != negate(gen[1]):
                        clauses += ['('+gen[0]+' v '+gen[1]+')']
                    else:
                        if
contradiction(goal, ('('+gen[0]+' v '+gen[1]+')'),
                            temp.append('('+gen[0]+' v '+gen[1]+')')
                            steps[''] = "Resolved {temp[i]}
{temp[j]} to {temp[-1]}, which is in turn null.
```



```
In A 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."
In A 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 = 'R v ~P  R v ~Q  ~R v P  ~R v Q' # (P^Q) <=> R.
        (R v ~P) v (R v ~Q) ^ (~R v P) ^ (~R v Q)
goal = 'R'
main(rules, goal)
```



Output:

Step	Clause	Derivation
1.	R v ~P	given.
2.	R v ~Q	given.
3.	~R v P	given.
4.	~R v Q	given.
5.	~R	Negated conclusion
6.		Resolved R v ~P and ~R v P to R v ~R, which is in turn null.

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

NLP
2/12/2024

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
```

Implement unification in FOL

Algo :-

function Unify (x, y, θ) returns a substitution to make x & y identical

if θ = failure then return false
else if x = y then return θ
else if variable ? (x) then return Unify-var (x, y, θ)
else if variable ? (y) then return Unify-var (y, x, θ)
else if compound ? (x) & compound ? (x, op, y, op, θ)
return unify (x, Args, y, Args, unify (x, op, y, op, θ))
else if List ? (x) & List ? (y) then.
return unify (x, Rest, y. Rest, Unify (x. First, y. First, θ))
else return failure.

function Unify-var (var, x, θ) return a substitution.
if {var/val} ∈ θ then return unify (val, x, θ)
else if {x/val} ∈ θ then return Unify (var, val, θ)
else if Occur-Check ? (var, x) then return failure
else return add {var/x} to θ.

Program-8

Unification

```
def unify_var(var, x, theta):
    """
    Helper function for unifying a variable with a term
    """
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta
def unify(x, y, theta={}):
    """
    Unify two expressions x and y with the given substitution theta
    """
    if theta is None:
        return None
    elif x == y:
        return theta
    elif isinstance(x, str) and x[0].islower():
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y[0].islower():
        return unify_var(y, x, theta)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return None
        for xi, yi in zip(x, y):
            theta = unify(xi, yi, theta)
            if theta is None:
```

```
        return None
    return theta
else:
    return None

# Example usage:
x = ['P', 'a', 'x']
y = ['P', 'y', 'z']
result = unify(x, y)
print(result)
# Sample input
expression1 = ['P', 'a', 'x']
expression2 = ['P', 'y', 'z']
# Unify the expressions
result = unify(expression1, expression2)
# Display the result
print("Input:")
print("Expression1:", expression1)
print("Expression2:", expression2)
print("\nOutput:")
if result is not None:
    print("Unification Successful!")
    print("Substitution theta:", result)
else:
    print("Unification Failed!")
```

```
Output:
107  exp1 = "Knows(A, x)"
108  exp2 = "Knows(y, z)"
109  substitutions = unify(exp1, exp2)
110  print("Substitutions:")
111  print(substitutions)
Substitutions:
[('A', 'y'), ('x', 'z')]
```

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)
```

Page

Q
Convert a given first logic statement to conjunctive Normal form (CNF)

Algo:-

1. Eliminate biconditional & implications
   - Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

2. move $\neg$ inwards.
   $\neg (\forall x\ p) \equiv \exists x\ \neg p$
   $\neg (\exists x\ p) \equiv \forall x\ \neg p$
   $\neg (\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$
   $\neg (\alpha \wedge \beta) \equiv \neg \alpha \vee \neg \beta$
   $\neg \neg \alpha \equiv \alpha$

3. Standardize variables apart by renaming them each quantifier should use a different variable

4. Skolemize: Each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variables.

5. Drop universal quantifiers.

6. Distribute $\wedge$ over $\vee$

Program-9

FOL to CNF

```
from sympy import symbols, to_cnf, pause_expr

def convert_to_cnf(logic_statement):
    # Parse the logic statement
    parsed_statement = parse_expr(logic_statement)

    # Convert to CNF
    cnf = to_cnf(parsed_statement)

    return cnf
if __name__ == "__main__":
    # Example: (A & B) | (~C & D)
    logic_statement = "(A & B) | (~C & D)"

    # Convert to CNF
    cnf_result = convert_to_cnf(logic_statement)

    print("Original Statement:", logic_statement)
    print("CNF Form:", cnf_result)
```

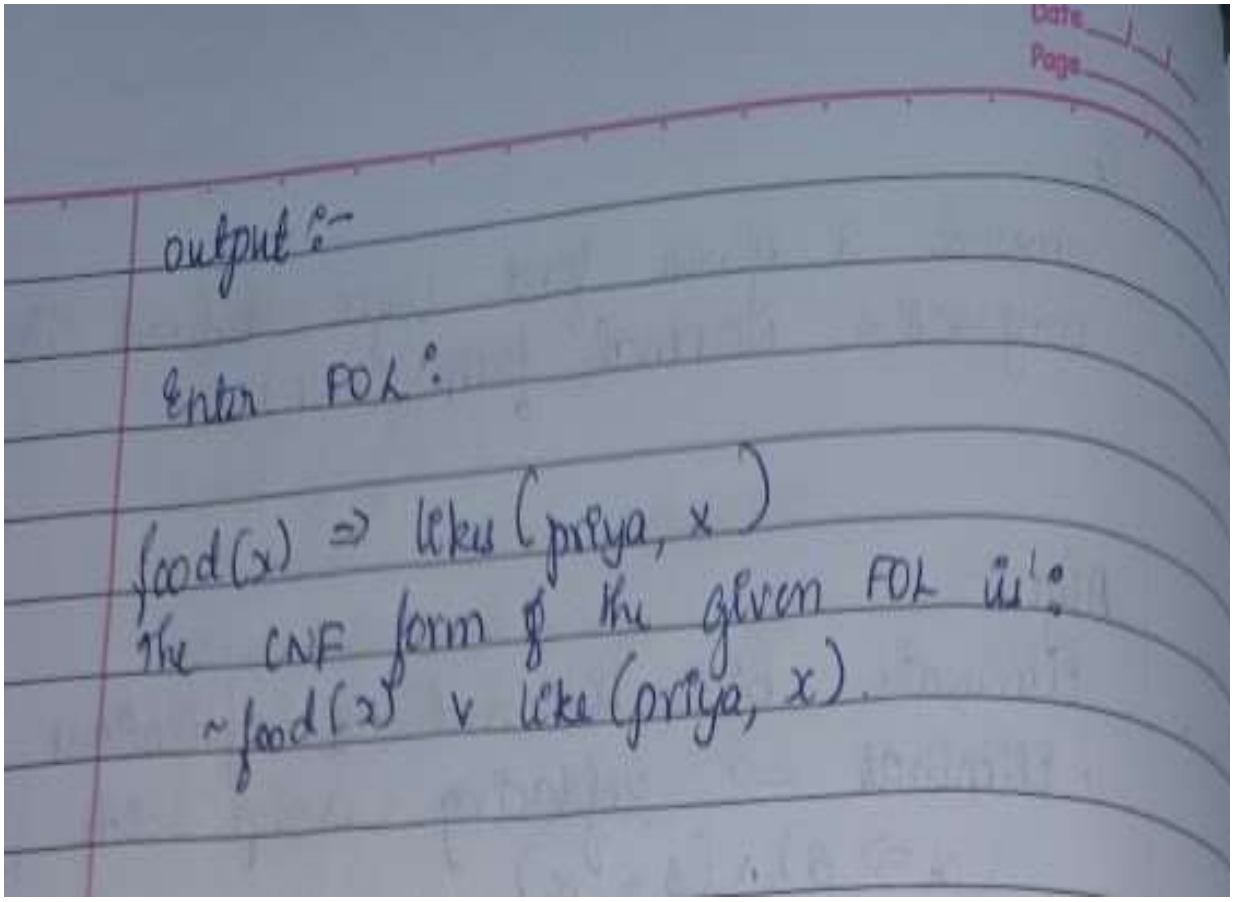
Output

3a print(fol_to_cnf("bird(x) => ~fly(x)"))

4a print(fol_to_cnf("∃x[bird(x) => ~fly(x)]"))

~bird(x) | ~fly(x)

[~bird(A) | ~fly(A)]


output :-

Enter FOL :

food(x) ⇒ likes(priya, x)

The CNF form of the given FOL is :

~food(x) ∨ likes(priya, 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)
```

Program - 10

Forward Reasoning

```
from sympy import symbols, Eq, And, Or, Implies, Not, satisfiable

# Define individuals (family members)
John, Mary, Alice, Bob = symbols('John Mary Alice Bob')

# Define predicates
Parent = symbols('Parent')
Grandparent = symbols('Grandparent')
# Define knowledge base
knowledge_base = [
    Eq(Parent(John, Alice), True),
    Eq(Parent(Mary, Alice), True),
    Eq(Parent(Alice, Bob), True),
    Implies(Parent(x, y), Grandparent(x, y)),
]

# Define query
query = Grandparent(John, Bob)

# Perform forward reasoning
def forward_reasoning(knowledge_base, query):
    new_facts = set()
    while True:
        for fact in knowledge_base:
            if ask(fact):
                continue
            if satisfiable(fact):
                new_facts.add(fact)
```

```
        if not new_facts:
            break

    knowledge_base.extend(new_facts)
return ask(query)
# Check if the query is true based on the knowledge base
result = forward_chaining(knowledge_base, query)

# Print the result
print("Query:", query)
print("Result:", result)
```

Output:

```
95  kb = KB()
96  kb.tell('missile(x) => weapon(x)')
97  kb.tell('missile(M1)')
98  kb.tell('enemy(x, America) => hostile(x)')
99  kb.tell('american(west)')
100 kb.tell('enemy(Nono, America)')
101 kb.tell('owns(Nono, M1)')
102 kb.tell('missile(x) & owns(Nono, x) => sells
                (west, x, Nono)')
103 kb.tell('american(x) & weapon(y) & sells
                (x, y, z) & hostile(z) => criminal(x)')
104 kb.query('criminal(x)')
105 kb.display()
```

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

All facts:
1. missile(M1)
2. weapon(M1)

3 enemy (Nona, America)
4. drink (Nona, M1)
5. hostile (Nona)
6. criminal (west)
7. American (west)
8. sells (west, M1, Nona)