VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

PRITHVI PRAKASH S (1BM21CS265)

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 **PRITHVI PRAKASH S (1BM21CS265),** 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.

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]:
```

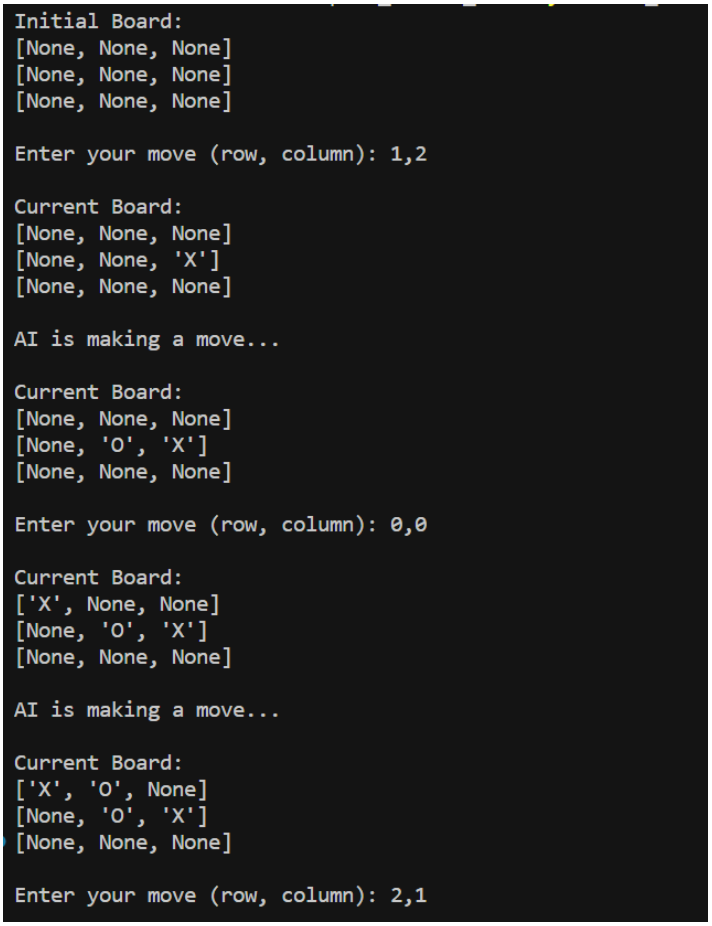
```
            s2.append(board[j][i])
        if (s2[0] == s2[1] == s2[2]):
            return s2[0]
    strikeD = []
    for i in [0, 1, 2]:
        strikeD.append(board[i][i])
    if (strikeD[0] == strikeD[1] == strikeD[2]):
        return strikeD[0]
    if (board[0][2] == board[1][1] == board[2][0]):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

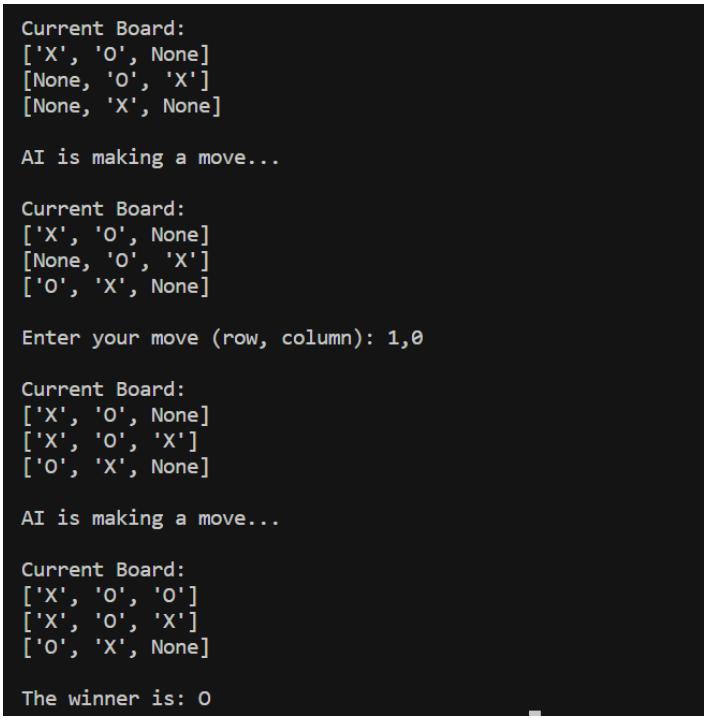
```
        return -1
    else:
        return 0


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:



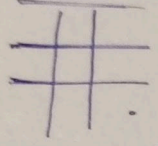


TIC TAC TOE - 2 (Using Minimax Algorithm)

Introduction

Rules
1. The game is played on a grid that's 3 squares by 3 squares.
2. You are X, your friend (opponent) is O. Players take turns putting their marks in empty squares.
3. First player to get 3 marks in a row (up, down, across or diagonally) is the winner & Tie
4. When all 9 squares are full Game is over.

Initial State



Goal State

Path Cost : 1 for each move

Search Strategy - Uses DFS (Depth First Search) We might not get solution always, hence the game ends in a tie.

DFS - Is not Complete

Python Code:

→ * Minimax Algorithm Back and forth b/w the 2 players, where the player whose "turn it is" desires to pick the move with maximum score

Aims to maximum payoff for the user and minimize the opponents payoff

Tic-Tac-Toe

def ConstBoard

print("Curr

for i in

use 1-D array

if (

// Current State of the Board iterate the elements (0,9)

Board → Clemp sp

→ 1 (

→ -1 (x).

def UserT

pos =

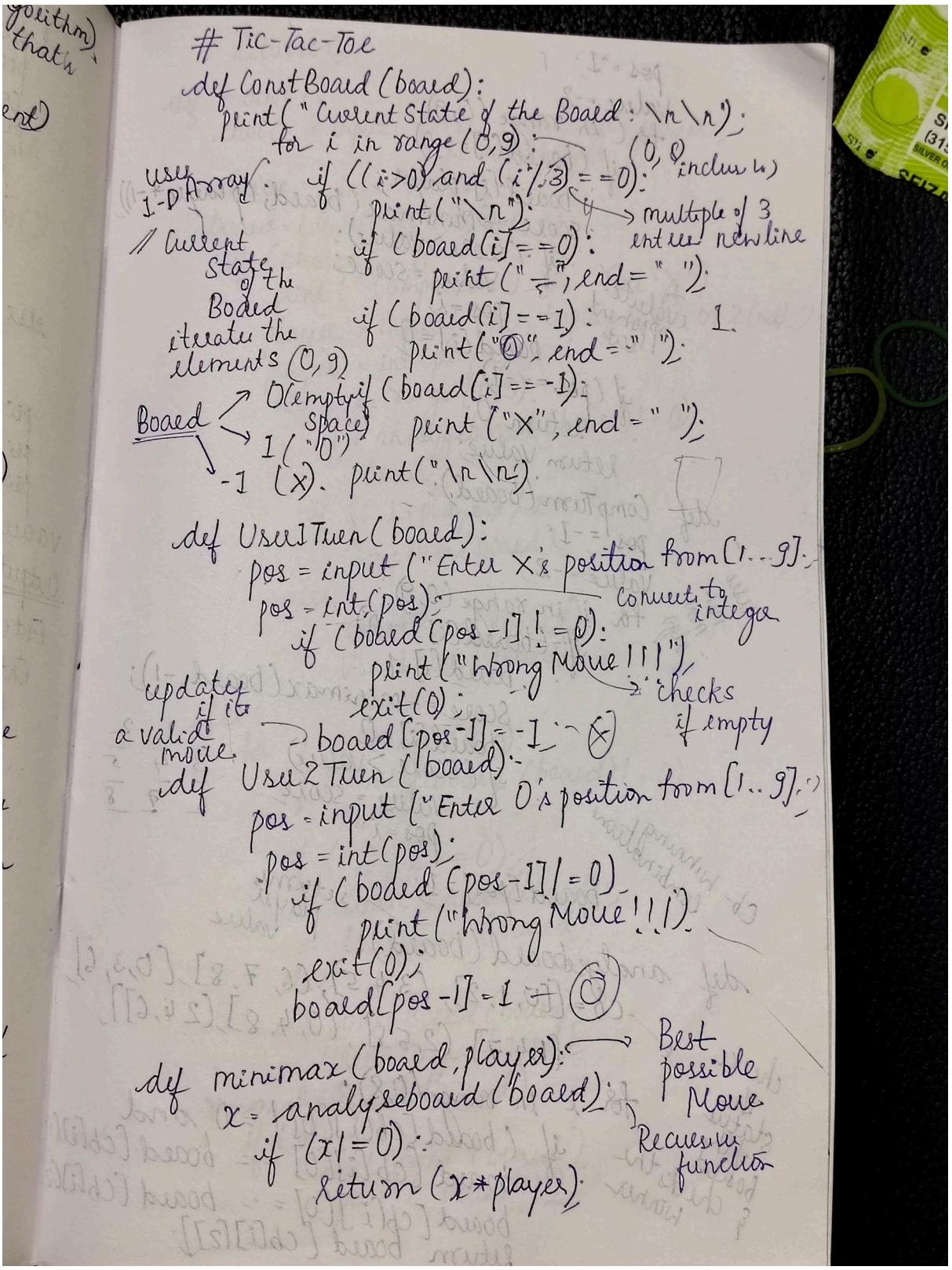
pos -

update if it a valid move

def Use

def mi

x


# Tic-Tac-Toe
def ConstBoard(board):
print("Current State of the 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("O", end=" ");
if(board[i]==-1):
print("X", end=" ");
print("\n\n");
def UserITurn(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=analyseboard(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
            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 analyseboard(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]];
```

to evaluate opponent score

use min max algo

cb - winning combinations

0 1 2
3 4 5
6 7 8

returns player value

checks the status of the board & checks the winner

```
return 0;
def main():
    choice = inp
    choice = in
    board = [0,0
    if (choic
        print(
        playe
        playe
        for i
        i
        el
else:
    for
x =
```

```
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 : 0 Vs You : X")
        player = input("Enter to play 1(st) or 2(nd):")
        player = int(player);
        for i in range(0,9):
            if(analyseboard(board)!=0);
                break;
            if((i+player)%2==0);
                CompTurn(board);
        else:
            ConstBoard(board);
            User1Turn(board);

    else:
        for i in range(0,9):
            if(analyseboard(board)!=0);
                break;
            if((i)%2==0);
                ConstBoard(board);
                User1Turn(board);
        else
            ConstBoard(board);
            User2Turn(board);
    x = analyseboard(board);
    if(x==0);
        ConstBoard(board);
        print("Draw!!!")
```

```
if (x==-1)
    const Board (board);
    print ("X Wins!!! O Lose!!!")
if (x==1)
    const Board (board)
    print ("X lose!!! O Wins!!!
main().
```

Output

Enter 1 for single player, 2 multiplayer 1

Computer O Vs You X

Enter to play 1(st) or 2(nd)

1

Current State

```
_ _ _
_ _ _
_ _ _
```

```
_ _ _
X _ _
_ _ _
```

Enter X position 1

```
X _ _
_ O _
_ _ _
```

Enter X position 3

```
X O X
_ O _
_ _ _
```

Enter X position 8

```
X O X
_ O _
O X _
```

Draw

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':
```

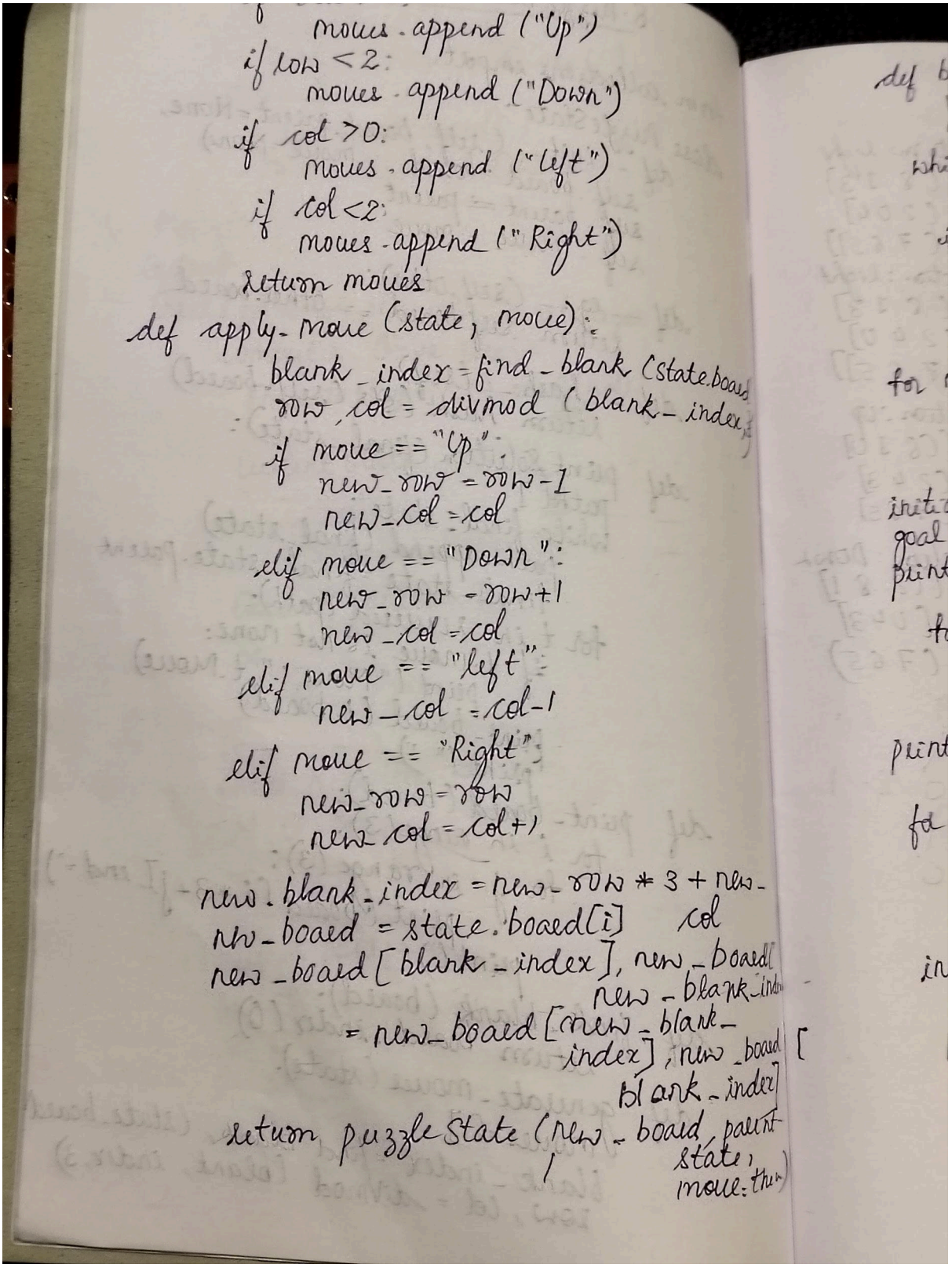
```
        temp[b-3],temp[b] = temp[b],temp[b-3]

    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
```

8-Puzzle Breadth First Search-3

```
from collections import
class PuzzleState:
    def __init__(self, board, parent=None,
        self.board = board    move=None)
        self.parent = parent
        self.move = move

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self)
        return hash(str(self.board)

    def print_solution(final_state):
        path=[]
        while final_state:
            path.append(final_state)
            final_state = final_state.parent
        for t in reversed(path):
            if t.move is not none:
                print("Move:", t.Move)
            print_board(t.board)
            print("")
def print_board(board):
    for i in range(3):
        for j in range(3):
            print(board[i*3+j], end=")
        print()
def find_blank(board):
    return board.index(0)
def generate_moves(state):
    moves=[]
    blank_index = find blank (state_board
    row, col = divmod(blank, index, 3)
```



```
        moves.append("Up")
    if row < 2:
        moves.append("Down")
    if col > 0:
        moves.append("Left")
    if col < 2:
        moves.append("Right")
    return moves
def apply_move(state, move):
    blank_index = find_blank(state.board)
    row, col = divmod(blank_index, 3)
    if move == "Up":
        new_row = row - 1
        new_col = col
    elif move == "Down":
        new_row = row + 1
        new_col = col
    elif move == "left":
        new_col = col - 1
    elif move == "Right":
        new_row = row
        new_col = col + 1

new_blank_index = new_row * 3 + new_
nw_board = state.board[i]    col
new_board[blank_index], new_board[
                    new_blank_index
    = new_board[new_blank_
                index], new_board[
                    blank_index]
return puzzleState(new_board, parent
                   state,
                   move: the)
```

```
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 : 0 Vs You : X")
        player = input("Enter to play 1(st) or 2(nd):")
        player = int(player);
        for i in range(0,9):
            if(analyseboard(board)!=0);
                break;
            if((i+player)%2==0);
                CompTurn(board);
        else:
            ConstBoard(board);
            User1Turn(board);

    else:
        for i in range(0,9):
            if(analyseboard(board)!=0);
                break;
            if((i)%2==0);
                ConstBoard(board);
                User1Turn(board);
        else
            ConstBoard(board);
            User2Turn(board);
    x = analyseboard(board);
    if(x==0);
        ConstBoard(board);
        print("Draw!!!")
```

```
if (x==-1)
    const Board (board);
    print ("X Wins!!! O Lose!!!")
if (x==1)
    const Board (board)
    print ("X Lose!!! O Wins!!!
main().
```

Output

Enter 1 for single player, 2 multiplayer 1

Computer O Vs You X

Enter to play 1(st) or 2(nd)

1

Current State

```
_ _ _
_ _ _
_ _ _
```

```
_ _ _
X _ _
_ _ _
```

Enter X position 1

```
X _ _
_ O _
_ _ _
```

Enter X position 3

```
X O X
_ O _
_ _ _
```

Enter X position 8

```
X O X
_ O _
O X _
```

Draw

3. Implement Iterative deepening search algorithm.

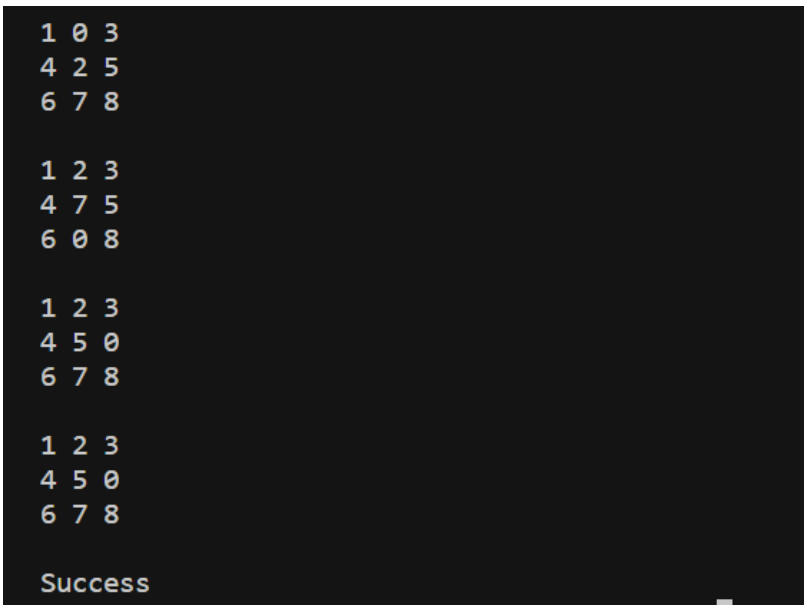
```
def iterative_deepening_search(src, target):
    depth_limit = 0
    while True:
        result = depth_limited_search(src, target, depth_limit, [])
        if result is not None:
            print("Success")
            return
        depth_limit += 1
        if depth_limit > 30:  # Set a reasonable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

```
def bfs(initial_state, goal_state):
    visited = set()
    queue = deque([initial_state])
    while queue:
        current_state = queue.popleft()
        if current_state == goal_state:
            print("Goal reached!")
            print_solution(current_state)
            return
        visited.add(current_state)
        for move, new_state in generate_moves(current_state):
            if new_state not in visited:
                queue.append((new_state, move))

initial_board = []
goal_board = []
print("Enter the initial Board Elements Based on Position")
for i in range(9):
    element = int(input(f"Element at position {i+1}:"))
    initial_board.append(element)
print("Enter the Goal Board Element Based on Position")
for i in range(9):
    element = int(input(f"Element at position {i+1}:"))
    goal_board.append(element)
initial_state = PuzzleState(initial_board)
goal_state = PuzzleState(goal_board)
bfs(initial_state, goal_state)
```

Output

Sol:

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

Goal State
[[2 8 1]
[0,4,3]
[7 6 5]]

Action: Up
[[1 0 3]
[8 2 4]
[7 6 5]]

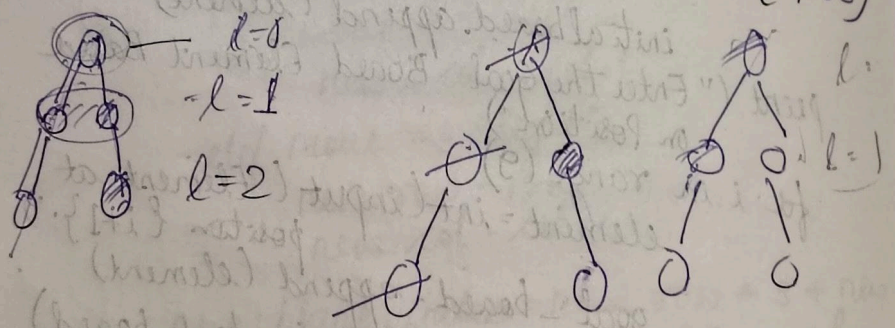
Action: Left
[[8 1 3]
[2 0 4]
[7 6 5]]

Action: Left
[[8 1 3]
[2 4 0]
[7 6 5]]

Action: Up
[[8 1 0]
[2 4 3]
[7 6 5]]

Action: Down
[[2 8 1]
[0 4 3]
[7 6 5]]

Action: Down
[[8 1 3]
[0 2 4]
[7 6 5]]



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)
```

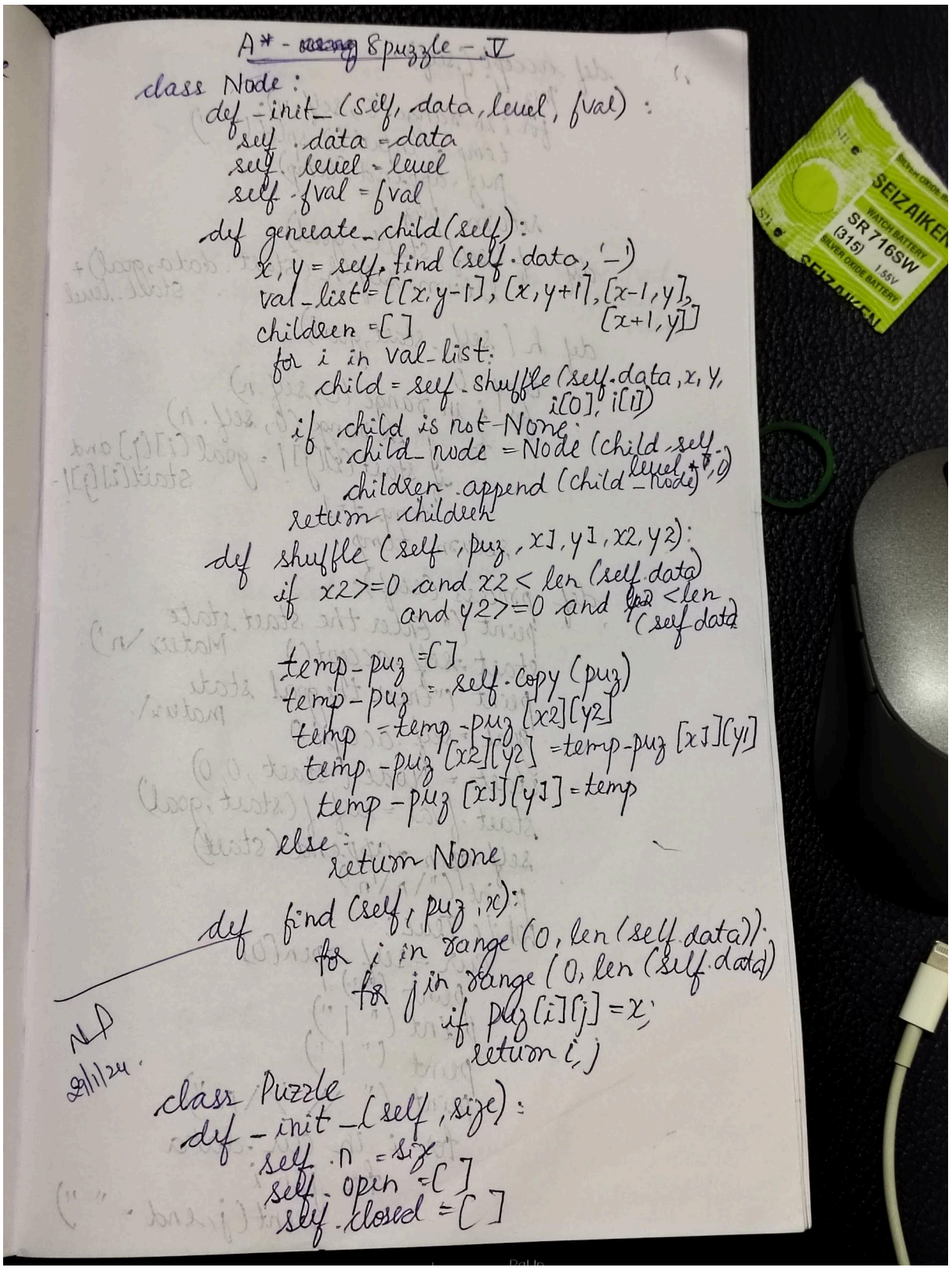
OUTPUT:

```
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
```

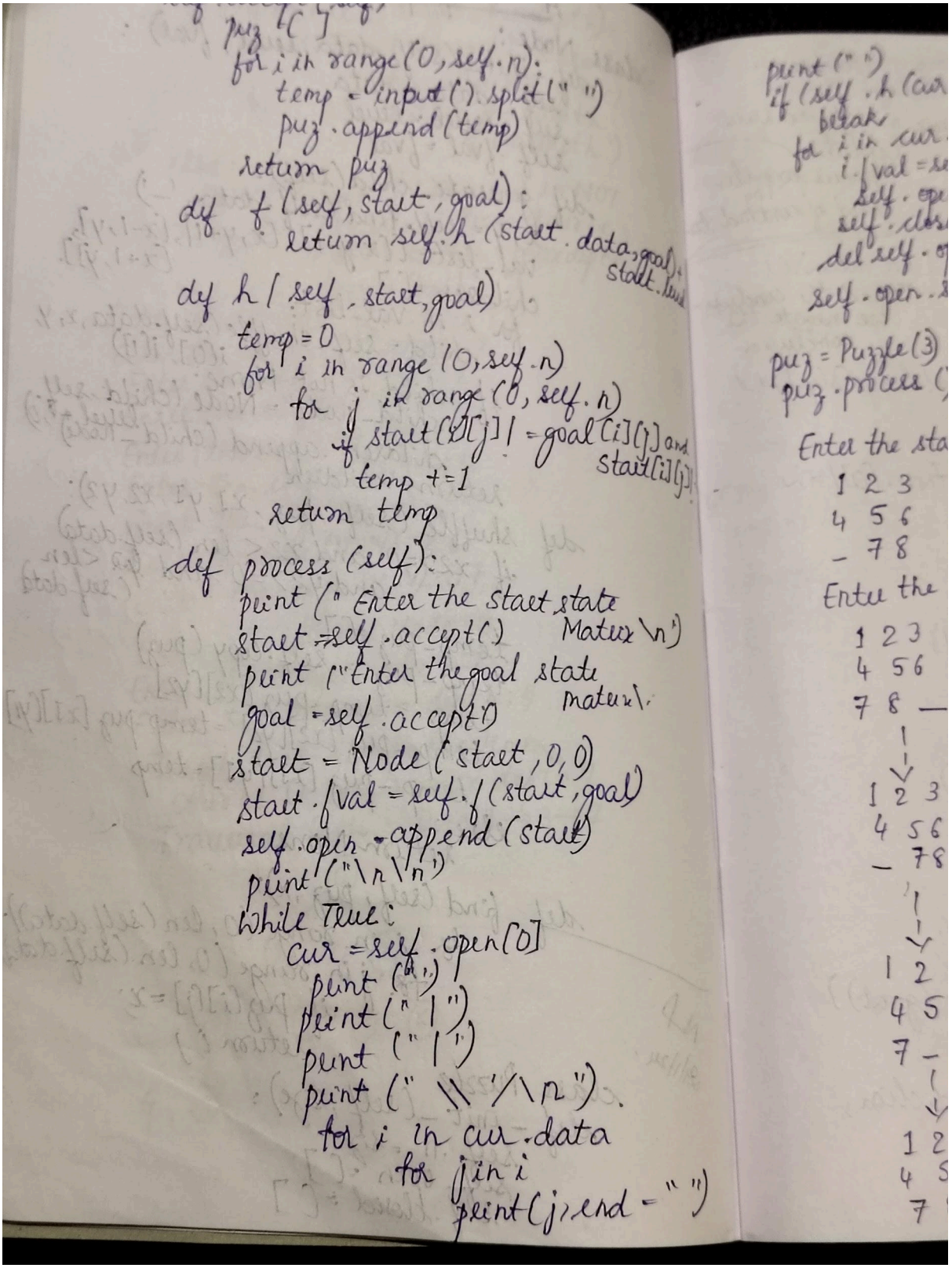
```
7 1 3
4 6 5
  2 8


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
```



A* - ~~misng~~ 8puzzle - IV

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval
    def generate_child(self):
        x, y = self.find(self.data, '_')
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level+1, 0)
                children.append(child_node)
        return children
    def shuffle(self, puz, x1, y1, x2, y2):
        if x2>=0 and x2 < len(self.data) and y2>=0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
        else:
            return None
    def find(self, puz, x):
        for i in range(0, len(self.data)):
            for j in range(0, len(self.data)):
                if puz[i][j] == x:
                    return i, j

class Puzzle
    def __init__(self, size):
        self.n = size
        self.open = []
        self.closed = []
```

29/1/24.



```
puz=[]
for i in range(0,self.n):
    temp=input().split(" ")
    puz.append(temp)
return puz
def f(self,start,goal):
    return self.h(start.data,goal)+start.level
def h(self,start,goal)
temp=0
for i in range(0,self.n)
    for j in range(0,self.n)
        if start[i][j]!=goal[i][j] and start[i][j]!=
            temp+=1
    return temp
def process(self):
print("Enter the start state Matrix\n")
start=self.accept()
print("Enter the goal state matrix\n")
goal=self.accept()
start=Node(start,0,0)
start.fval=self.f(start,goal)
self.open.append(start)
print("\n\n")
While True:
    cur=self.open[0]
    print(" ")
    print(" | ")
    print(" | ")
    print(" \\ | / \n")
    for i in cur.data
        for j in i
            print(j,end=" ")
```

```
print(" ")
if(self.h(cur
    break
for i in cur
    i.fval=se
    self.ope
    self.clos
    del self.o
    self.open.s
puz=Puzzle(3)
puz.process()
Enter the sta
1 2 3
4 5 6
- 7 8
Enter the
1 2 3
4 5 6
7 8 -
1
1 2 3
4 5 6
- 7 8
1
1 2
4 5
7 -
1
1 2
4 5
7
```

```
print(" ")
if (self.h(cur.data, goal) == 0);
    break
for i in cur.generate_child():
    i.fval = self.f(i, goal)
    self.open.append(i)
self.closed.append(cur)
del self.open[0]
self.open.sort(key = lambda x: x.fval,
                reverse = False)

puz = Puzzle(3)
puz.process()
```

Enter the start state matrix

1 2 3
4 5 6
_ 7 8

Enter the Goal State Matrix

1 2 3
4 5 6
7 8 _

|
↓

1 2 3
4 5 6
_ 7 8

|
↓

1 2 3
4 5 6
7 _ 8

|
↓

1 2 3
4 5 6

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
```

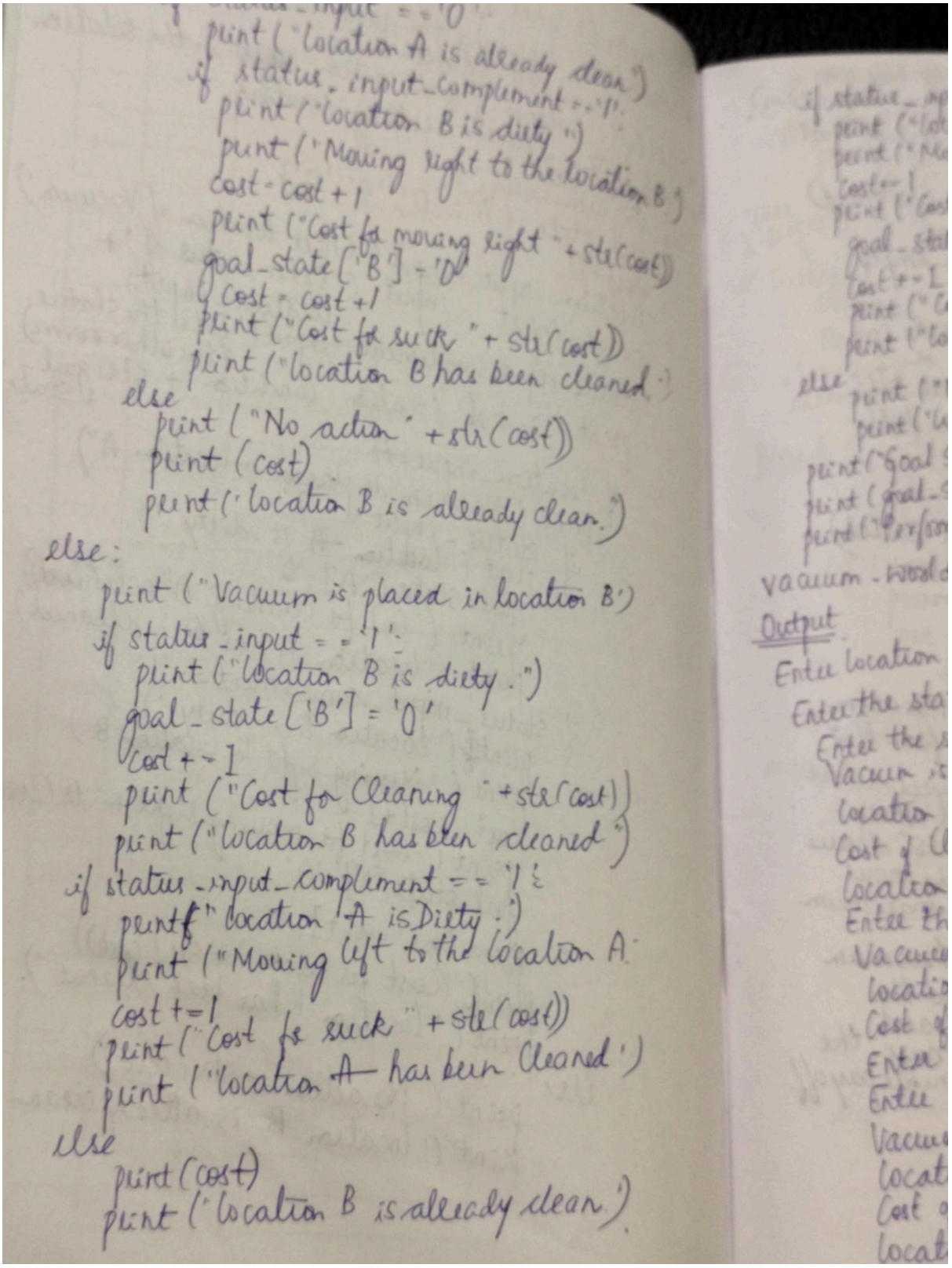
```
 1    0   >0<   0
 0    0    0    0
 0    0    0    0

 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
```

Vacuum Cleaner Problem - 1

* Search Cost: How long the agent takes to come up with solution
* Path Cost: Expense of each action in the solution
* Goal State: Desired resulting condition

Python Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of Vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter the status of the other room")
    print("Initial location Condition" + str(goal_state))

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



```
        print("location A is already clean")
        if status_input_complement == '1':
            print("location B is dirty")
            print("Moving right to the location B")
            cost = cost + 1
            print("Cost for moving right" + str(cost))
            goal_state['B'] = '0'
            cost = cost + 1
            print("Cost for suck" + str(cost))
            print("location B has been cleaned")
    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':
        printf("location A is Dirty.")
        print("Moving left to the location A")
        cost += 1
        print("Cost for suck" + str(cost))
        print("location A has been Cleaned")
    else
        print(cost)
        print("location B is already clean.")
```

Output

```
        if status_input_complement == '1':
            print ("location A is dirty.")
            print ("Moving left to the location A.")
            cost += 1
            print ("Cost for moving left " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print ("Cost for suck " + str(cost))
            print ("location A has been cleaned.")
        else
            print ("No action " + str(cost))
            print ("location A is already clean")
    print("Goal State:")
    print (goal_state)
    print ("Performance Measurement :" + str(cost))
vacuum_world()
```

Output

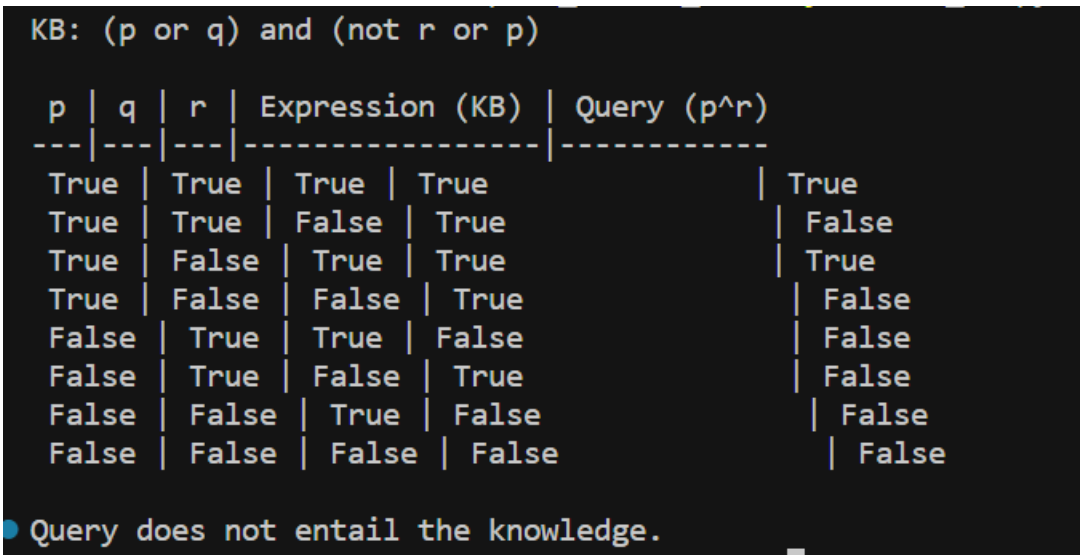
Enter location of Vacuum A
Enter the status of A 1
Enter the status of other room 1
Vacuum is placed in location A
location A is dirty
Cost of Cleaning A 1
location A has been cleaned
Enter the status of other room 0.
Vacuum is placed in location B.
location B is clean.
Cost of Cleaning B 0
Enter the location of Vacuum A
Enter the status of A 0.
Vacuum is placed in location A
location A is clean.
Cost of Cleaning A 0
location A has been cleaned.

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:


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.

KnowledgeBase Entailment

```
variable = { "p": 0, "q": 1, "r": 2 }
priority = { "~": 3, "V": 1, "^": 2 }

def eval(i, val1, val2):
    if i == "^":
        return val2 and val1
    return val2 or val1

def isOperand(c):
    return c.isalpha() and c != "V"

def isLeftParanthesis(c):
    return c == "("

def isRightParanthesis(c):
    return c == ")"

def isEmpty(stack):
    return len(stack) == 0

def peek(stack)
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False

def toPostfix(infix):
    stack = []
    postfix = ""
    for c in infix
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c)
                stack.append(c)
            elif isRightParanthesis(c):
```

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

Contradiction is found when ~R is assumed true.
Hence R is true.

C
S
-A
S → -B
C ∧ -B → A

CNF
-S v -B
C ∧ -B v A
C
S
-A

Entered NLP 29/1/24.

(1) C
(2) -S v -B
(3) -C v B v A
(4) S
(5) -A
(6) B v A (1) & (3)
(7) -B (2) & (4)
(8) A (6) & 7

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.
```

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

Make use of inference rule Resolution (proves proposition by proof of contradiction)

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

To find the conclusion we negate the conclusion and add it to the given statements

deductive inference

negated statement is false then original must be true

CNF

$a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$

$(a \rightarrow b) = \neg a \vee b$

$\neg a$

```
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
```

negated statement is false then original must be true

CNF
$a \leftrightarrow b = (a \rightarrow b) \wedge (b \rightarrow a)$

$(a \rightarrow b) = \neg a \vee b$

deduction inference

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

Contradiction is found when ~R is assumed true
Hence R is true

C
S
-A
S → -B
C ∧ -B → A

CNF
-S v -B
C ∧ -B v A
C
S
-A

(1) C
(2) -S v -B
(3) -C v B v A
(4) S
(5) -A
(6) B v A (1) & (3)
(7) -B (2) & (4)
(8) A (6) & 7

Entered
NLP
29/1/24.

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
```

Unification of First Order Logic

```
def unify(expr1, expr2):
    func1, args1 = expr1.split('(', 1)
    func2, arg2 = expr2.split('(', 1)
    if func11 = func2:
        print("Expression cannot be unified
              Different function.")
        return None
    args1 = args1.rstrip(')').split(',')
    args2 = arg2.rstrip()).split(',')
    substitution = {}
    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower() and
                                a11=a2.
        substitution[a1] =a2
        elif not a1.islower() and a2.islower():
        substitution[a2] =a1
    elif a11 =a2:
        print("Not")
        return None
    return substitution
def apply-substitution
    for key, value in substitution.items():
        expr = expr.replace(key, value)
    return expr
if __name__ == "__main__")
    expr1 = input = (Enter the first')
    expr2 = input ("Enter the second").
    substitution = unify(expr1, expr2)
if substitution
    print ("The substitution are:')
```

print("Unified expression: ", expr2.result)

Output

Expression 1 : Knows(x, John)
Expression 2 : Knows(Smith, y)
[('Smith', 'x'), ('John', 'y')]

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)
```

First Order logic to CNF

```
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 = getPredicate(expression)[0]
        params = getAttributes(expression)[0].
        strip('()').split(',')
        return [predicate, para

class KB:
    def __init__(self):
        self.facts = set()
        self.implication = set()
    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
& sells(x, y, z) & hostile(z) => criminal(y)
kb.query('criminal(x)')
kb.display()
```

Output

FOL Statement $\forall x\ (P(x) \Rightarrow Q(x))$

FOL converted to CNF

$[\sim P(A) \mid Q(A)]$

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):
```

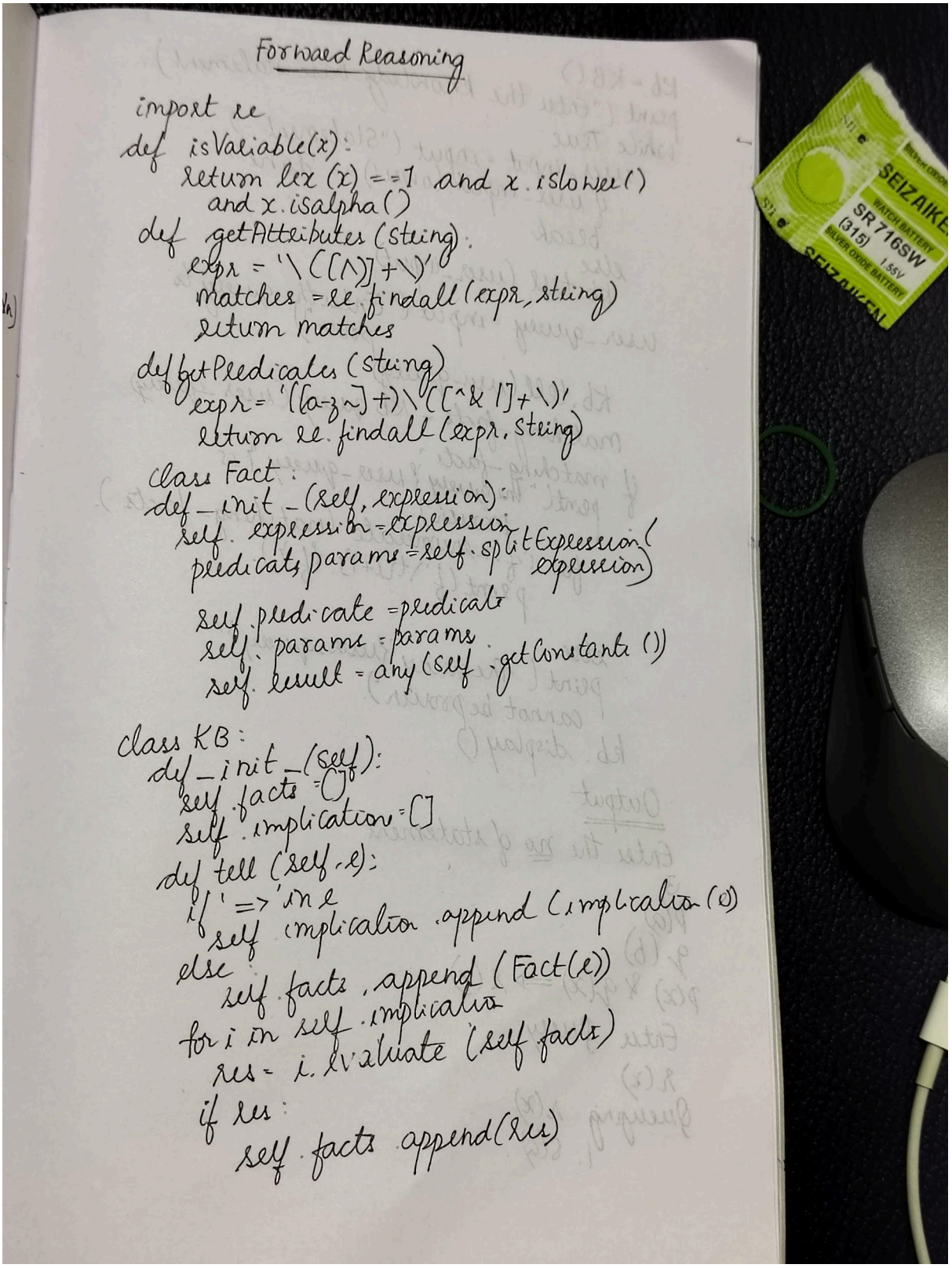
```
        return self.result

    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)
```



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())

class KB:
  def __init__(self):
    self.facts = []
    self.implication = []
  def tell(self, e):
    if '=>' in e
      self.implication.append(Implication(e))
    else:
      self.facts.append(Fact(e))
    for i in self.implication
      res = i.evaluate(self.facts)
      if res:
        self.facts.append(res)
```

```
kb = KB()
print("Enter the knowledge base statement).
while True:
    user_input = input("Statement :)
    if user_input.lower() == 'done':
        break
    else
        kb.tell(user_input)
user_query = input("Enter the query to
                    proove")
    kb.tell(user_query)
    matching_facts = kb.query(user_query)
if matching_facts:
    print("The query {user_query} is
                proven")
    for i,f in enumerate(matching_facts):
        print(f'\t{i+1}. {f}')

    else
    print("The query {user_query}
        cannot be proven")
    kb.display()
```

Output

Enter the no of statement

3

P(a)

q(b)

p(x) & q(x) => r(x)

Enter Query

r(x)

Querying r(x)

1. s(a).