VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

ARTIFICIAL INTELLIGENCE

Submitted by

IMRAN WADRALI (1BM21CS077)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Oct 2023-Feb 2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled **"ARTIFICIAL INTELLIGENCE"** carried out by **IMRAN WADRALI (1BM21CS077)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of Artificial Intelligence Lab - **(22CS5PCAIN)**work prescribed for the said degree.

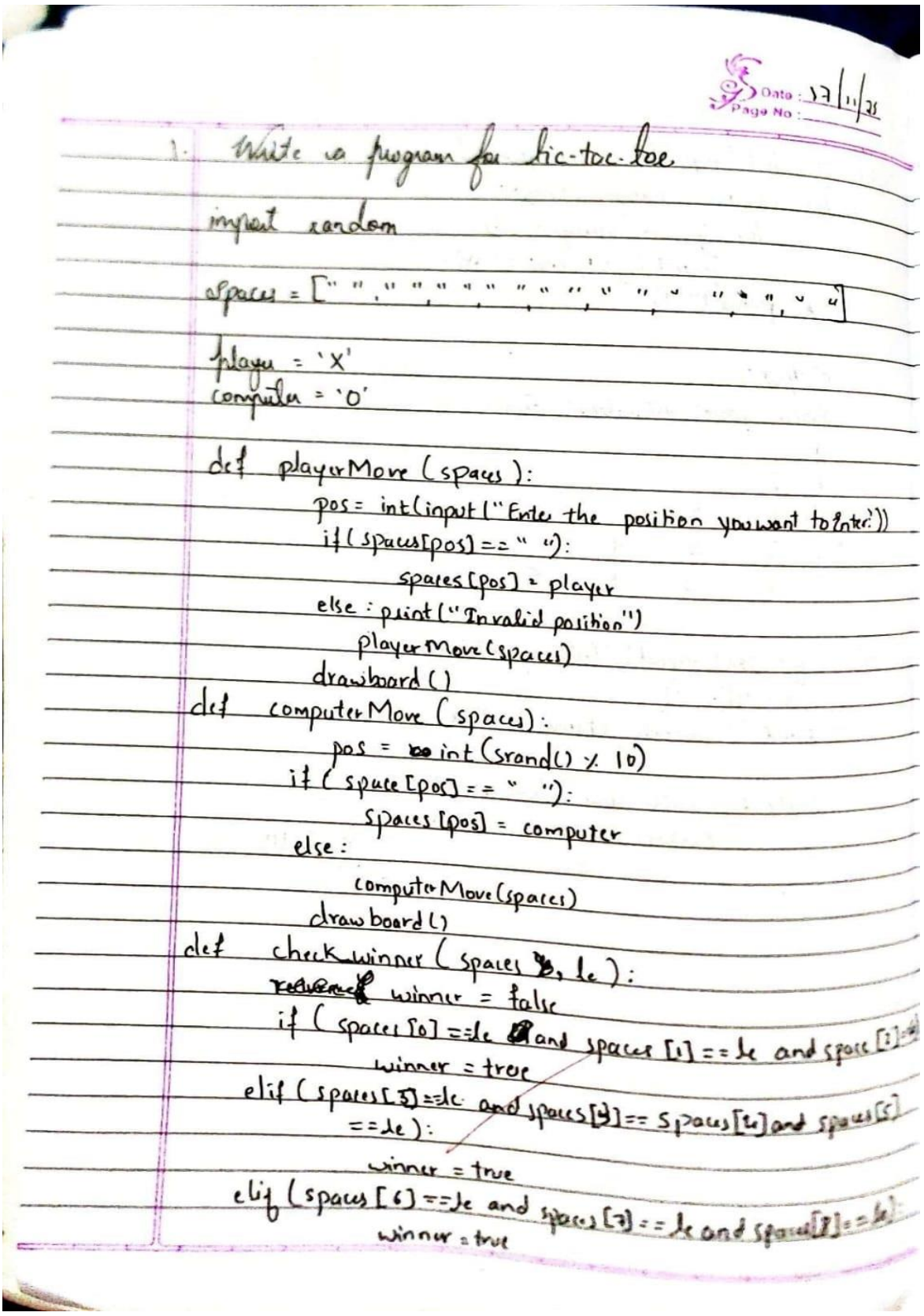
Swathi Sridharan
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Table of Contents

SL No	Name of Experiment	Page No
1	Implement Tic –Tac –Toe Game	1-7
2	Implement 8 puzzle problem	8-12
3	Implement Iterative deepening search algorithm.	13-17
4	Implement A* search algorithm.	18-23
5	Implement vaccum cleaner agent.	24-30
6	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .	31-34
7	Create a knowledge base using prepositional logic and prove the given query using resolution	35-40
8	Implement unification in first order logic	41-48
9	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	49-56
10	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	57-63

1.Implement Tic –Tac –Toe Game.



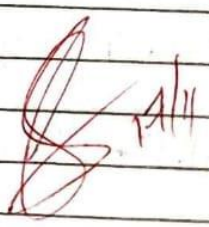
1. Write a program for tic-tac-toe

```
import random

spaces = [" ", " ", " ", " ", " ", " ", " ", " ", " ", " "]

player = 'X'
computer = 'O'

def playerMove (spaces):
    pos = int(input("Enter the position you want to enter:"))
    if (spaces[pos] == " "):
        spaces [pos] = player
    else : print ("Invalid position")
        playerMove (spaces)
    drawboard ()
def computerMove (spaces):
    pos = int (srand() % 10)
    if ( space [pos] == " "):
        spaces [pos] = computer
    else:
        computerMove (spaces)
    drawboard ()
def check_winner ( spaces, le ):
    winner = false
    if ( spaces [0] == le and spaces [1] == le and spaces [2] ==
        winner = true
    elif ( spaces [3] == le and spaces [4] == spaces [5] and spaces [5]
        == le ):
        winner = true
    elif (spaces [6] == le and spaces [7] == le and spaces [8] == le):
        winner = true
```

main()

Algorithm:

Step 1: make a spaces of 9 elements.
Step 2: Take input player move
Step 3: Check winner (if win stop the game)
Step 4: Take computer's random input
Step 5: Check winner. (if won stop the game)



```
elif(spaces[0] == le and spaces[3] == le and spaces[6] == le):
        winner = true
elif(spaces[1] == le and spaces[4] == le and spaces[7] == le):
        winner = true
elif(spaces[2] == le and spaces[5] == le and spaces[8] == le):
        winner = true
elif(spaces[0] == le and spaces[4] == le and spaces[8] == le):
        winner = true
elif(spaces[2] == le and spaces[4] == le and spaces[6] == le):
        winner = true
return winner

def drawboard():
    print(spaces[0] + "|" + spaces[1] + "|" + spaces[2])
    print("-----")
    print(spaces[3] + "|" + spaces[4] + "|" + spaces[5])
    print("-----")
    print(spaces[6] + "|" + spaces[7] + "|" + spaces[8])

def main():
    running = true
    drawboard()
    while(running):
        playermove(spaces)
        if(checkwinner(spaces, player)):
            running = false
            break()
        computerMove(spaces)
        if(checkwinner(spaces, computer)):
            running = false
            break()
```

```
tic=[]
import random
def board(tic):
    for i in range(0,9,3):
        print("+"+"-"*29+"+")
        print("|"+" "*9+"|"+" "*9+"|"+" "*9+"|")
        print("|"+" "*3,tic[0+i],," "*3+"|"+" "*3,tic[1+i],," "*3+"|"+" "*3,tic[2+i],," "*3+"|")
        print("|"+" "*9+"|"+" "*9+"|"+" "*9+"|")
    print("+"+"-"*29+"+")



def update_comp():
    global tic,num
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='X'
            if winner(num-1)==False:
                #reverse the change
                tic[num-1]=num
            else:
                return
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='O'
            if winner(num-1)==True:
                tic[num-1]='X'
                return
            else:
```

```
            tic[num-1]=num
        num=random.randint(1,9)
    while num  not in tic:
        num=random.randint(1,9)
    else:
        tic[num-1]='X'



def update_user():
    global tic,num
    num=int(input("enter a number on the board :"))
    while num not in tic:
        num=int(input("enter a number on the board :"))
    else:
        tic[num-1]='O'

def winner(num):
    if tic[0]==tic[4] and tic[4]==tic[8] or tic[2]==tic[4] and tic[4]==tic[6]:
        return True
    if tic[num]==tic[num-3] and tic[num-3]==tic[num-6]:
        return True
    if tic[num//3*3]==tic[num//3*3+1] and tic[num//3*3+1]==tic[num//3*3+2]:
        return True
    return False



try:
    for i in range(1,10):
        tic.append(i)
    count=0
```

```
    #print(tic)
    board(tic)
    while count!=9:
        if count%2==0:
            print("computer's turn :")
            update_comp()
            board(tic)
            count+=1
        else:
            print("Your turn :")
            update_user()
            board(tic)
            count+=1
        if count>=5:
            if winner(num-1):
                print("winner is ",tic[num-1])
                break
            else:
                continue
except:
    print("\nerror\n")
```

OUTPUT:

```
Imran-1BM21CS077
```

```
+-------------------------------+
Your turn :
enter a number on the board :4
+-------------------------------+
|       |       |       |
|   1   |   X   |   3   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   O   |   5   |   6   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------------------------------+
computer's turn :
+-------------------------------+
|       |       |       |
|   X   |   X   |   3   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   O   |   5   |   6   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------------------------------+
Your turn :
enter a number on the board :5
+-------------------------------+
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
+-------------------------------+
|       |       |       |
|   1   |   2   |   3   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   4   |   5   |   6   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------------------------------+
computer's turn :
+-------------------------------+
|       |       |       |
|   1   |   X   |   3   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   4   |   5   |   6   |
|       |       |       |
+-------------------------------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------------------------------+
Your turn :
enter a number on the board :4
```

2 .Solve 8 puzzle problems.

Date 24/11/23
Page No

Lab-3 : 8- puzzle

1) Define the state representation: Represent the initial & final states of the puzzle.
like
initial = [1, 2, 3, 4, 5, 6, 7, 0, 8]
Goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

2) Initialize the Queue: Begin by making a Queue and store the states. Add the initial state of the puzzle to the queue.

3) Make rules for Moving tiles: The empty tile can move only in 4 directions.
up, down, left, right.
It should take one step at a time
Divide rules like

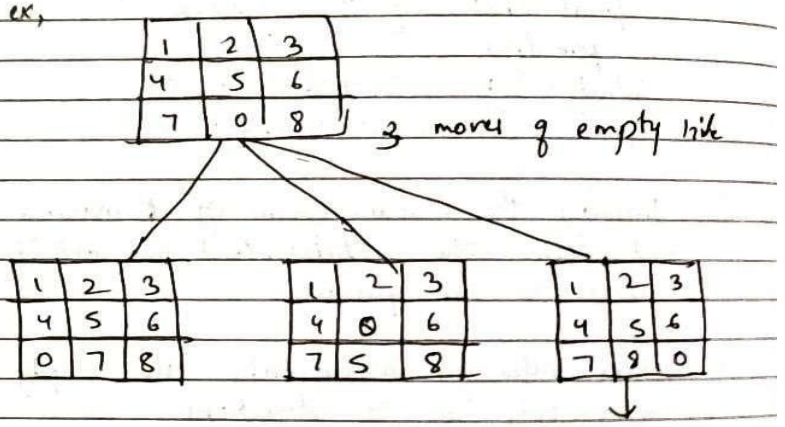
2	3	2
3	4	3
2	3	2

→ Like the corner elements can move only in 2 directions
→ The centre element can move in all 4 directions
→ The corner middle elements can move in 3 directions

4) Next, every time load the current state of board and create the possible moves until current state matches goal state and make a path for the same. Also ensure the states are not already visited to avoid cycles.

5] If BFS reach a solution give the path of solution else print no solution.

For ex,



Program:

```
from collections import deque

def bfs_8puzzle(initial_state, goal_state):
    queue = deque([[initial_state, []]])
    visited = set()
    while queue:
        current_state, path = queue.popleft()
    if (current_state == goal_state): return path
```

Output.

Solution:

[(1, 2, 3, 4, 0, 6, 7, 5, 8)
(1, 2, 3, 4, 5, 6, 7, 0, 8)
(1, 2, 3, 4, 5, 6, 7, 8, 0)]

```
def bfs(src,target):
    queue=[]
    queue.append(src)
    exp=[]
    while len(queue)>0:
        source=queue.pop(0)
        #print("queue",queue)
        exp.append(source)

        print(source[0],'|',source[1],'|',source[2])
        print(source[3],'|',source[4],'|',source[5])
        print(source[6],'|',source[7],'|',source[8])
        print("-----------")
        if source==target:
            print("Success")
            return
        poss_moves_to_do=[]
        poss_moves_to_do=possible_moves(source,exp)
        #print("possible moves",poss_moves_to_do)
        for move in poss_moves_to_do:
            if move not in exp and move not in queue:
              #print("move",move)
              queue.append(move)

def possible_moves(state,visited_states):
    b=state.index(0)

    #direction array
    d=[]
    if b not in [0,1,2]:
```

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


    pos_moves_it_can=[]


    for i in d:
        pos_moves_it_can.append(gen(state,i,b))
    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in
visited_states]


def gen(state,m,b):
    temp=state.copy()
    if m=='d':
        temp[b+3],temp[b]=temp[b],temp[b+3]
    if m=='u':
        temp[b-3],temp[b]=temp[b],temp[b-3]
    if m=='l':
        temp[b-1],temp[b]=temp[b],temp[b-1]
    if m=='r':
        temp[b+1],temp[b]=temp[b],temp[b+1]
    return temp


src=[1,2,3,4,5,6,0,7,8]
target=[1,2,3,4,5,6,7,8,0]
bfs(src,target)
```

OUTPUT

```
Imran-1BM21CS077
```

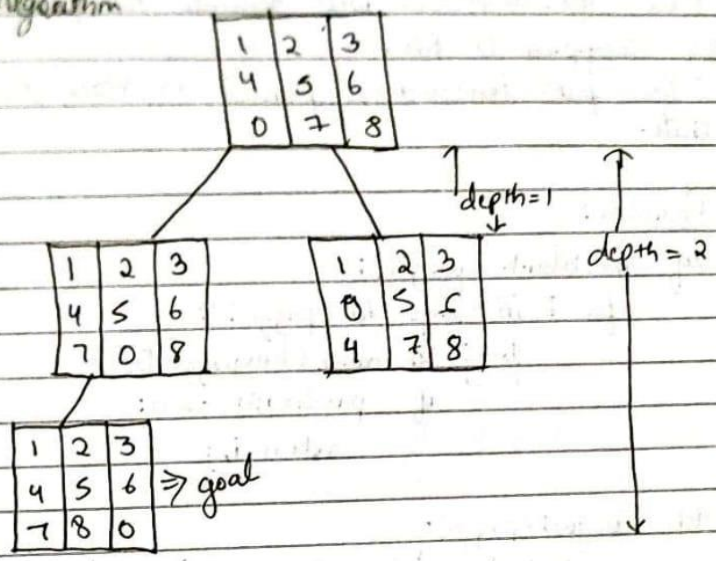
```
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8
-----------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
-----------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0
-----------
Success
```

3. Implement Iterative deepening search algorithm.

8/12/23

8-puzzle iterative deepening search algorithm

Algorithm



Algorithm:

1] Initialize the initial state = [] and goal state for the 8puzzle

goal_state = [1,2,3,4,5,6,7,8,0] #0 is empty tile

2] Set the depth = 1 and expand the initial state. The depth_limited_search(depth) is performed

```
if node.state = goal
    return node
else
    for neighbour in getneighnode(state)
        child = puzzlenode(neighbour, node)
        result = depth_limited_search(depth-1)
    if result == true;
        return result;
```

3) After one iteration when depth = 1 increment the depth by 1 and perform limited search again
4) Here get_neighbour will generate the possible moves by swapping '0' tile.
5) The path traversed is printed to reach the goal state.

Program:

```
def find_blank (puzzle):
    for i in range (len (puzzle)):
        for j in range (len(puzzle[i])):
            if puzzle[i][j] == 0:
                return i, j

def is_goal (puzzle):
    goal_state = [[1,2,3],[4,5,6],[7,8,0]]
    return puzzle == goal_state

def move_tile (puzzle, direction):
    blank_row, blank_col = find_blank (puzzle)
    new_puzzle = [row[:] for row in puzzle]

    if direction == 'up' and blank_row > 0:
        new_puzzle [blank_row][blank_col], new_puzzle
        [blank_row-1][blank_col] = new_puzzle[row...
            [blank_col], new_puzzle [blank_row...
    elif direction == 'down' and blank_row < 2:
        new_puzzle [blank_row][blank_col], new_puzzle
        [blank_row-1][blank_col], new_puzzle
```

```
def id_dfs(puzzle, goal, get_moves):
    import itertools
#get_moves -> possible_moves
    def dfs(route, depth):
        if depth == 0:
            return
        if route[-1] == goal:
            return route
        for move in get_moves(route[-1]):
            if move not in route:
                next_route = dfs(route + [move], depth - 1)
                if next_route:
                    return next_route

    for depth in itertools.count():
        route = dfs([puzzle], depth)
        if route:
            return route



def possible_moves(state):
    b = state.index(0)  # ) indicates White space -> so b has index of it.
    d = []  # direction
    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
```

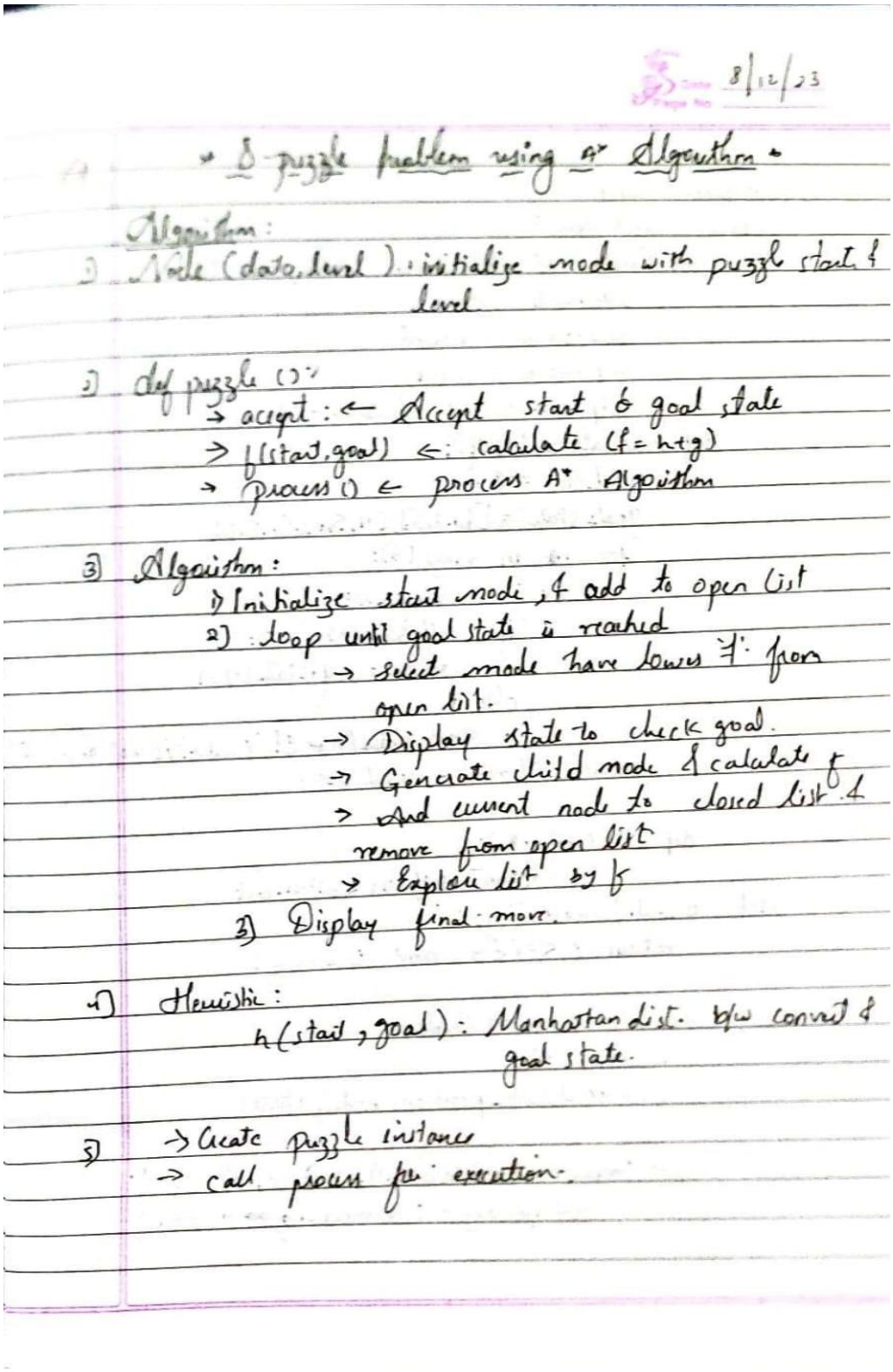
```
        d.append('r')

    pos_moves = []
    for i in d:
        pos_moves.append(generate(state, i, b))
    return pos_moves


def generate(state, m, b):
    temp = state.copy()

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

    return temp


# calling ID-DFS
initial = [1, 2, 3, 0, 4, 6, 7, 5, 8]
goal = [1, 2, 3, 4, 5, 6, 7, 8, 0]

route = id_dfs(initial, goal, possible_moves)

if route:
```

```
    print("Success!! It is possible to solve 8 Puzzle problem")
    print("Path:", route)
else:
    print("Failed to find a solution")
```

OUTPUT

```
Imran-1BM21CS077


Level: 0

               1 2 5
               3 4 _
               6 7 8

Level: 1

               1 2 _
               3 4 5
               6 7 8

Level: 2

               1 _ 2
               3 4 5
               6 7 8

Level: 3

               _ 1 2
               3 4 5
               6 7 8

Success
```

4. Implement A* search algorithm.



8/12/23

* 8 puzzle problem using A* Algorithm *

Algorithm:

1) Node (data, level): initialize node with puzzle state & level

2) def puzzle ():
   → accept: ← Accept start & goal state
   → f(start, goal) ← calculate (f = h+g)
   → process () ← process A* Algorithm

3) Algorithm:
   1) Initialize start node, & add to open list
   2) loop until goal state is reached
      → select node have lowest f from open list.
      → Display state to check goal.
      → Generate child node & calculate f
      → Add current node to closed list & remove from open list
      → Explore list by f
   3) Display final move.

4) Heuristic:
   h(start, goal): Manhattan dist. b/w current & goal state.

5) → Create puzzle instance
   → call process for execution.

class

```
Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value are out
            of limits the return None """
```

```
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None

def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j



class Puzzle:
    def __init__(self,size):
```

```
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp



    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
```

```
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\/ \n")
            for i in cur.data:
                for j in i:
                        print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)
```

puz = Puzzle(3)

puz.processs

OUTPUT

```
Imran-1BM21CS077
```

```
Enter the start state matrix

1 2 3
4 5 6
_ 7 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _



  |
  |
 \'/

1 2 3
4 5 6
_ 7 8

  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |

   _

   |
   |
  \'/

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

5. Implement vaccum cleaner agent.

Date: 22/12/23
Page No:

* Vacuum Cleaner Problem *

Algorithm:

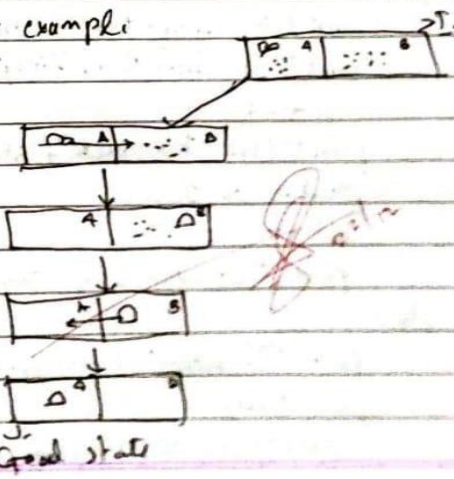
1] Initalize the starting and goal state; the goal state begin being to have both rooms clean.

2] If status = Dirty then clean
else if location = A & status = clean return right
else if location = B & status = clean return left
else exit.

3] Based on above rules, take user input on cleanliness status and show the actions of vacuum cleaner.

4] Once both rooms are clean the vacuum cleaner has accomplished the task and reached goal state and then exits.

For example:



Code:

```
def vacuum_world():
    goal_state = {'A': 0, 'B': 0}
    cost = 0

    location_input = input("Enter location of vacuum")
    status_input = int(input("Enter status of" + location_input))
    status_input_complement = int(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
            cost += 1
            print("Cost of CLEANING A" + str(cost))
            print("Loc A has been cleared")

            if status_input_complement == 1:
                print("Moving right to location B")
                print("Location B is Dirty")
                cost += 1
                print("Cost for suck" + str(cost))
                goal_state['B'] = 0
                cost += 1
                print("Location B is cleaned")
            else:
                print("No action" + str(cost))
                print(cost)
                print("Loc B is already clean")
```

```
else:
    print("Vaccum is placed in location B")
    if status_input == 1
        print("Location B is Dirty.")
        good_state['B'] = 0
        cost += 1
        print("COST for CLEANING" + str(cost))
        print("loc B has been cleaned")

        if status_input_complement = = 1
            print("Moving LEFT to locA")
            print("Location A is dirty")

            cost += 1
            print("COST of moving LEFT" + str(cost))
            good_state['A'] = 0
            cost += 1
            print("Cost of suck" + str(cost))
            print("Loc A has been cleaned")
        else:
            print("No action" + str(cost))
            print("Loc A is already clean")

print("GOAL STATE")
print(good_state)
print("performance measurement" + str(cost))

vaccum world()
```

```
def vacuum_world():
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

    location_input = input("Enter Location of Vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")

    if location_input == 'A':
        # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            # suck the dirt  and mark it as clean
            cost += 1                    #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1                    #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                cost += 1                    #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
```

```
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")
        if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':# if B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1                    #cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # suck the dirt and mark it as clean
            cost += 1                    #cost for suck
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            # suck and mark clean
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    # Location B is Dirty.
    if status_input == '1':
        print("Location B is Dirty.")
        # suck the dirt  and mark it as clean
        cost += 1  # cost for suck
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")
```

```
    if status_input_complement == '1':
        # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1  # cost for moving right
        print("COST for moving LEFT" + str(cost))
        # suck the dirt and mark it as clean
        cost += 1  # cost for suck
        print("COST for SUCK " + str(cost))
        print("Location A has been Cleaned.")

else:
    print(cost)
    # suck and mark clean
    print("Location B is already clean.")

    if status_input_complement == '1':  # if A is Dirty
        print("Location A is Dirty.")
        print("Moving LEFT to the Location A. ")
        cost += 1  # cost for moving right
        print("COST for moving LEFT " + str(cost))
        # suck the dirt and mark it as clean
        cost += 1  # cost for suck
        print("Cost for SUCK " + str(cost))
        print("Location A has been Cleaned. ")
    else:
        print("No action " + str(cost))
        # suck and mark clean
        print("Location A is already clean.")
```

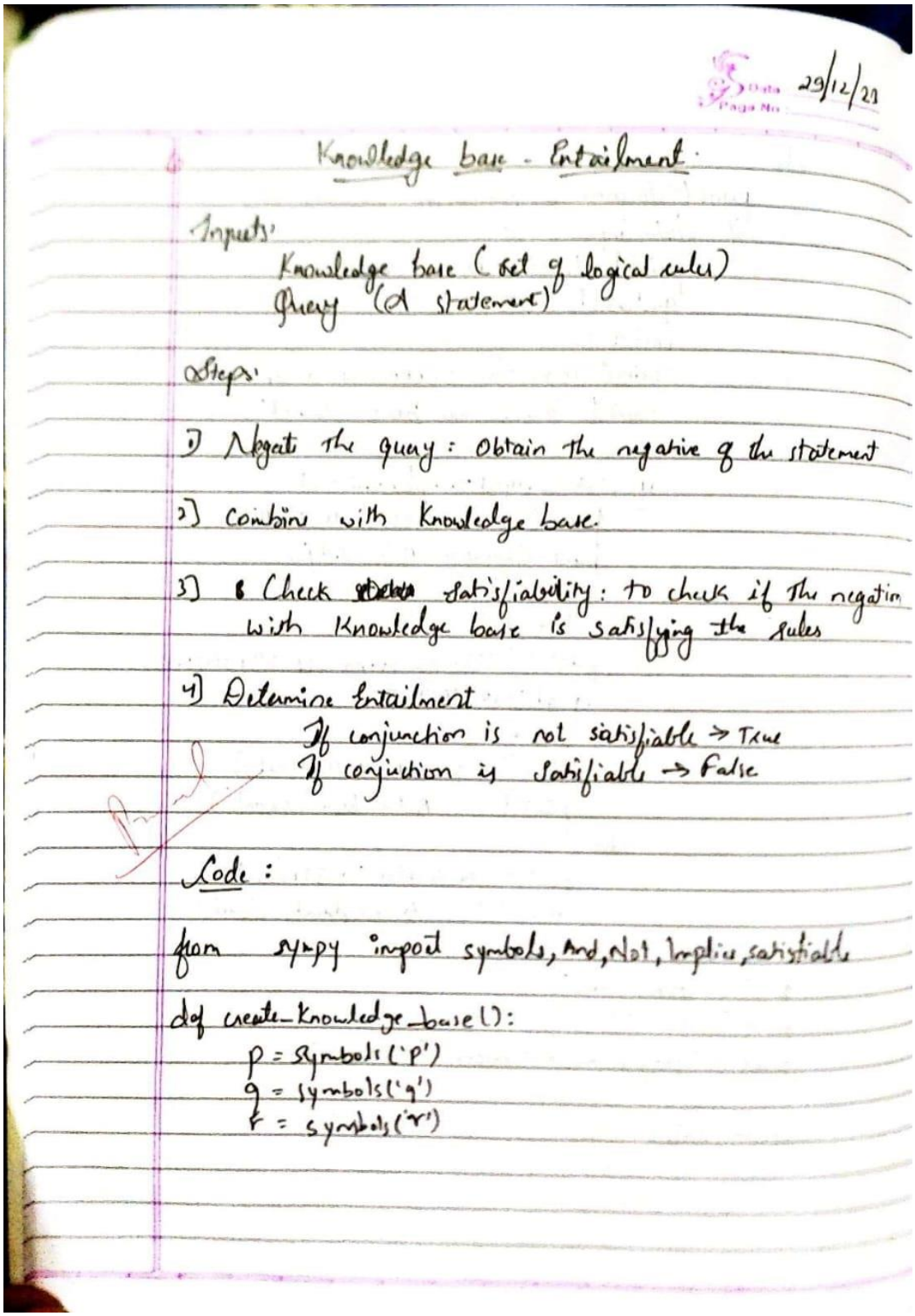
```
    # done cleaning
    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

print("0 indicates clean and 1 indicates dirty")
vacuum_world()
```

OUTPUT:

```
Imran-1BM21CS077
```

```
0 indicates clean and 1 indicates dirty
Enter Location of Vacuumb
Enter status of b1
Enter status of other room1
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for SUCK 3
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

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



Date 29/12/21
Page No

Knowledge base - Entailment

Inputs:
Knowledge base (set of logical rules)
Query (A statement)

Steps:

1) Negate the query: Obtain the negative of the statement
2) Combine with Knowledge base.
3) Check satisfiability: to check if the negation with Knowledge base is satisfying the rules
4) Determine Entailment
If conjunction is not satisfiable → True
If conjuction is satisfiable → False

Code :

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

def create_knowledge_base():
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')
```

```
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_knowledgebase()

    query = symbols('not p')

    result = query_entails(kb, query)

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

Output

Knowledge Base : ~r & (implies(p,q)) & (implies(q,r))

Query : (not, p)

Query entails Knowledge base : False

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

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

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

    return knowledge_base

def query_entails(knowledge_base, query):
    # Check if the knowledge base entails the query
    entailment = satisfiable(And(knowledge_base, Not(query)))

    # If there is no satisfying assignment, then the query is entailed
    return not entailment

if __name__ == "__main__":
    # Create the knowledge base
    kb = create_knowledge_base()

    # Define a query
    query = symbols('p')
```

```
# Check if the query entails the knowledge base
result = query_entails(kb, query)

# Display the results
print("Knowledge Base:", kb)
print("Query:", query)
print("Query entails Knowledge Base:", result)
```

OUTPUT:

```
Enter the knowledge base: (p^q)v(~pvq)
Enter the query: pvq
[True, True, True] :kb= True :q= True
[True, True, False] :kb= True :q= True
[True, False, True] :kb= False :q= True
[True, False, False] :kb= False :q= True
[False, True, True] :kb= True :q= True
[False, True, False] :kb= True :q= True
[False, False, True] :kb= True :q= False
Doesn't entail!!
```

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

Knowledge-Base. Propositional Logic

1. Code 2:

```
def negate_literal(literal)
    if literal[0] == '~':
        return literal[1:]
    else:
        return '~' + literal

def resolve(C1, C2):
    resolved_clause = set(C1) | set(C2)
    for literal in C1:
        if negate_literal(literal) in C2:
            resolved_clause.remove(literal)
            resolved_clause.remove(negate_literal(literal))
    return tuple(resolved_clause)

def resolution(knowledge_base):

    while True:
        new_clauses = set()
        for i, C1 in enumerate(knowledge_base)
            for j, C2 in enumerate(knowledge_base)

                if i != j:
                    newclause = resolve(C1, C2)
                if len(new_clause) > 0 & new_clause
                        not in knowledge_base:
                    new_clause.add(new_clause)

        if not new_clause
            break.
```

```
        Knowledge_base != new_clause
        Return Knowledge_base

if __name__ == '__main__':
        Kb = [('p', b), ('~p', 'r'), ('~q', '~r')]
        result = resolution(Kb)
        print('original Kb', Kb)
        print('Resolved Kb', result)
```

Output:

Enter Statement = negation

The Statement not entitled in Knowledge base

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1


def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''


def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms
```

split_terms('~PvR')

OUTPUT:

```
Enter the clauses separated by a space: p v ~q ~r v p ~q
Enter the query: ~p
Trying to prove (p)^(v)^(~q)^(~r)^(v)^(p)^(~q)^(~p) by contradiction....
Knowledge Base entails the query, proved by resolution
```

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


def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
```

```
        for c in terms1:
            if negate(c) in terms2:
                t1 = [t for t in terms1 if t != c]
                t2 = [t for t in terms2 if t != negate(c)]
                gen = t1 + t2
                if len(gen) == 2:
                    if gen[0] != negate(gen[1]):
                        clauses += [f'{gen[0]}v{gen[1]}']
                    else:
                        if contradiction(goal,f'{gen[0]}v{gen[1]}'):
                            temp.append(f'{gen[0]}v{gen[1]}')
                            steps[''] = f'Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in
turn null. \
                            \nA contradiction is found when {negate(goal)} is assumed as true.
Hence, {goal} is true."
                            return steps
        elif len(gen) == 1:
                    clauses += [f'{gen[0]}']
                else:
                    if contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                        temp.append(f'{terms1[0]}v{terms2[0]}')
                        steps[''] = f'Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in
turn null. \
                        \nA contradiction is found when {negate(goal)} is assumed as true. Hence,
{goal} is true."
                        return steps
        for clause in clauses:
            if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
                temp.append(clause)
                steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
        j = (j + 1) % n
    i += 1
```

return steps

rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)

goal = 'R'

main(rules, goal)

```
Step    |Clause |Derivation
------------------------------
 1.     | 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.
```

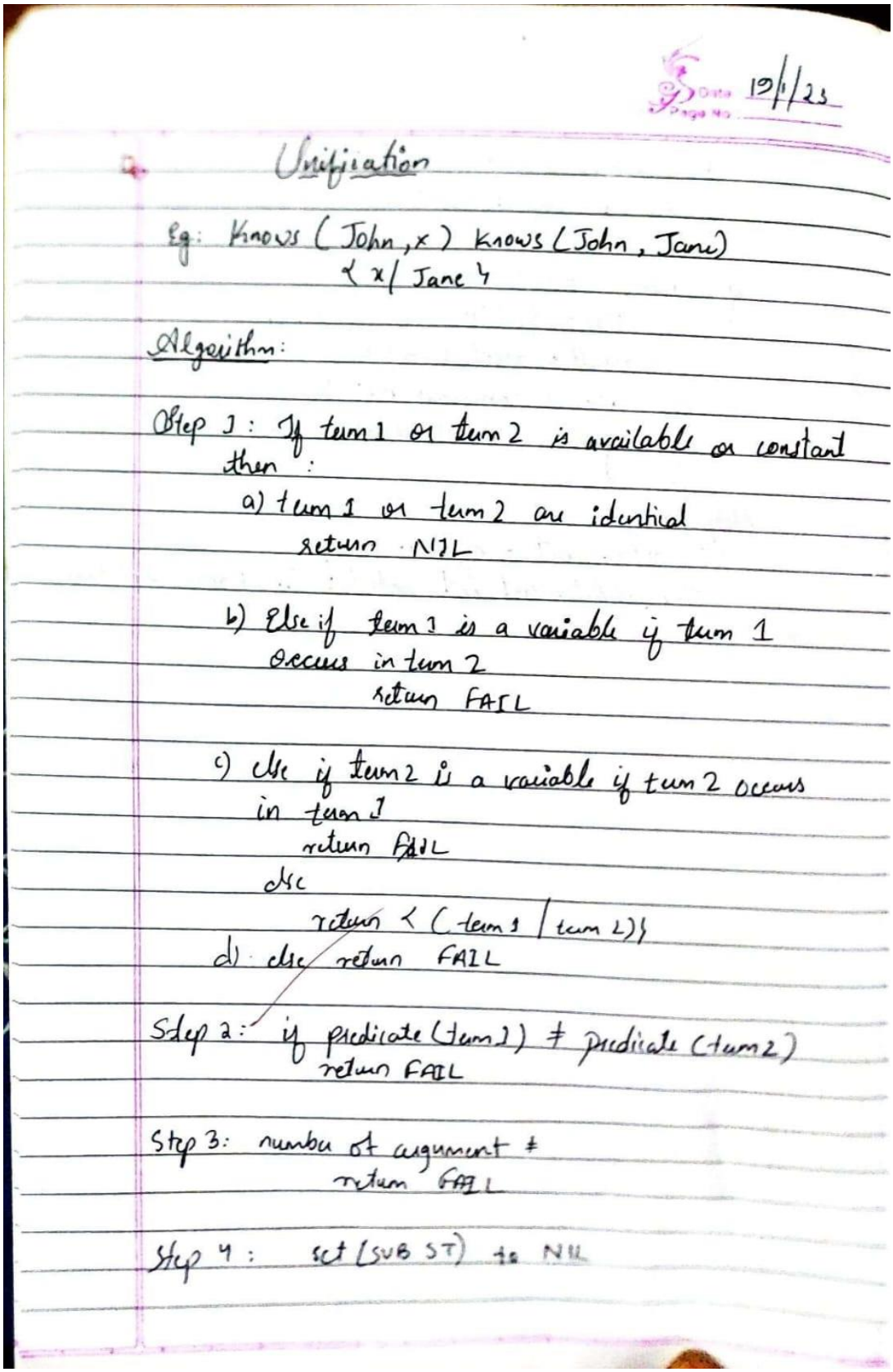
rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR

goal = 'R'

main(rules, goal)

```
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.
```

8. Implement unification in first order logic :



Unification

Eg: Knows (John, x) Knows (John, Jane)
{x/ Jane}

Algorithm:

Step 1: If term 1 or term 2 is variable or constant then:
a) term 1 or term 2 are identical
return NIL

b) Else if term 1 is a variable if term 1 occurs in term 2
return FAIL

c) else if term 2 is a variable if term 2 occurs in term 1
return FAIL
else
return {(term 1 / term 2)}
d) else return FAIL

Step 2: if predicate (term 1) ≠ predicate (term 2)
return FAIL

Step 3: number of argument ≠
return FAIL

Step 4: set (SUB ST) to NIL

Step 5: for i=1 to the number of elements in term 1

a) call unify (ith term 1, ith term 2)
put result into S
S = FAIL
return FAIL.

b) if S ≠ NIL
a) Apply S to the remainder of both L_1 & L_2
b) SUBST. APPEND (S, SUBST)

step 6: Return SUBST

Code:

```
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 isVariable(char):
    return char.islower() and len(char)==1
```

```
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 unify(exp1, exp2):
    if exp1 == exp2
        return []

    if isConstant(exp1):
        return [(exp1, exp2)]

    if isVariable(exp1):
        if checkOccurs(exp1, exp2):
            return false
        else:
            return [(exp2, exp1)]

    if initialSubstitution != []:
        tail1 = apply(tail1, initial)
        tail2 = apply(tail1, initial)
```

initial Substitution extend (renaming sub)
return initialSubstitution.

Output:

Substitutions:
[('A','Y'), ('Y','X')]

Substitutions:
[('A','Y'), ('Mother(Y)','X')]

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?

def getInitialPredicate(expression):
    return expression.split("(")[0]

def isConstant(char):
    return char.isupper() and len(char) == 1

def isVariable(char):
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    attributes = getAttributes(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
```

```
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


def getFirstPart(expression):
    attributes = getAttributes(expression)
    return attributes[0]


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression


def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False

    if isConstant(exp1):
        return [(exp1, exp2)]
```

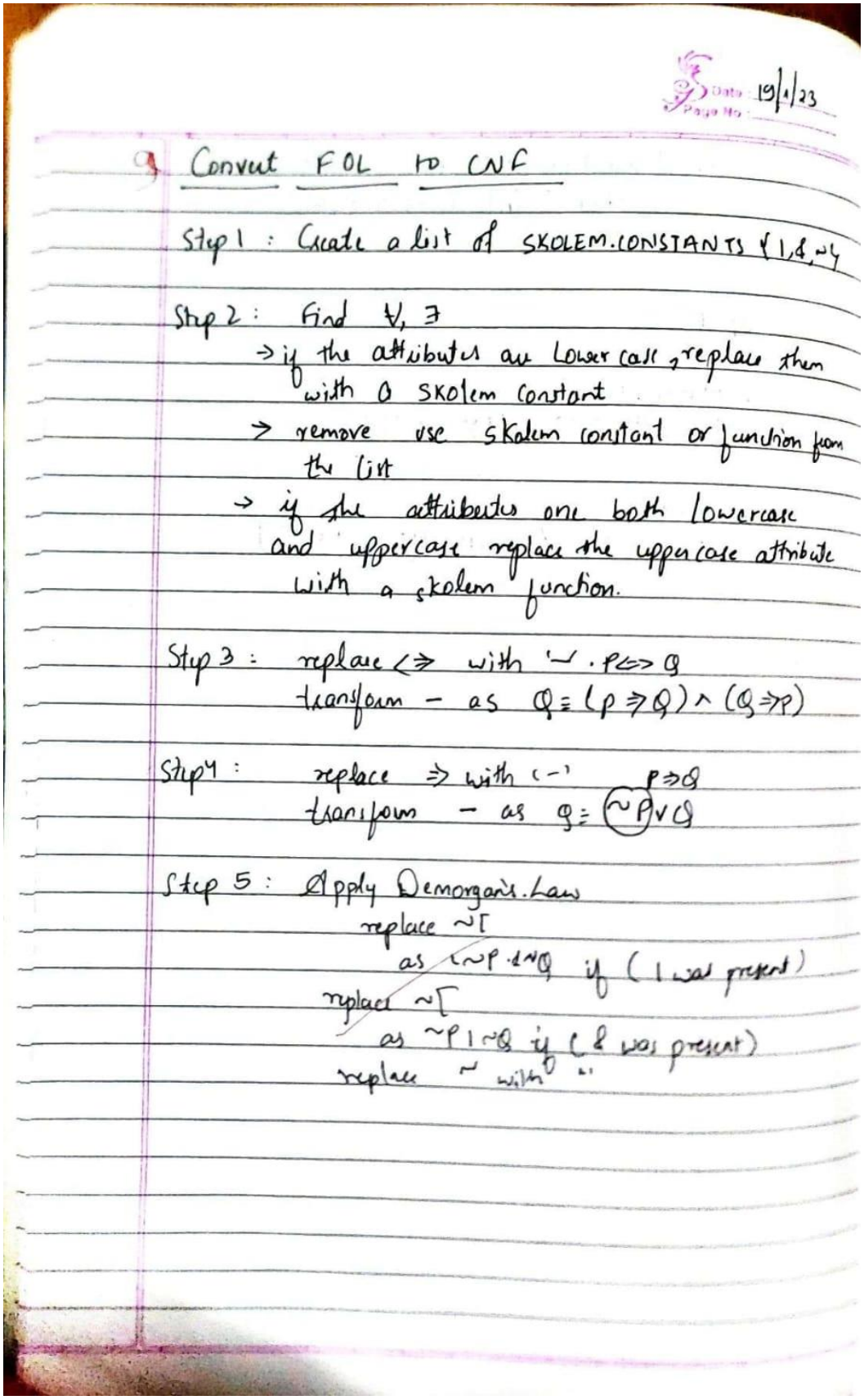
```
if isConstant(exp2):
    return [(exp2, exp1)]

if isVariable(exp1):
    if checkOccurs(exp1, exp2):
        return False
     else:
        return [(exp2, exp1)]

if isVariable(exp2):
    if checkOccurs(exp2, exp1):
        return False
    else:
        return [(exp1, exp2)]

if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Cannot be unified")
    return False

attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False

head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)
initialSubstitution = unify(head1, head2)
if not initialSubstitution:
    return False
if attributeCount1 == 1:
```

```
        return initialSubstitution

    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    if initialSubstitution != []:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)
remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return False

    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution


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

OUTPUT

```
Expression1: parent(x, y)
Expression2: parent(john, mary)
Predicates do not match. Cannot be unified
Substitutions:
False
```

9.Convert a given first order logic statement into Conjunctive Normal Form (CNF).



Date 19/1/23
Page No.

Convert FOL to CNF

Step 1 : Create a list of SKOLEM.CONSTANTS {1,2,...}

Step 2 : Find ∀, ∃
→ if the attributes are Lower case, replace them with a skolem constant
→ remove use skolem constant or function from the list
→ if the attributes are both lowercase and uppercase replace the uppercase attribute with a skolem function.

Step 3 : replace <⇒ with '~' . P<=>Q
transform - as Q≡(P⇒Q)∧(Q⇒P)

Step 4 : replace ⇒ with (~) P⇒Q
transform - as Q≡(~P∨Q)

Step 5 : Apply Demorgan's Law
replace ~[
as (~P.∨Q if (was present)
replace ~[
as ~P|~Q if (& was present)
replace ~ with ..

Code:

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findAll(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getpredicate(string):
    expr = '[a-zA-Z]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def deMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '~' in string

    for predicate in getPredicate(string):
        string = string.replace(predicate, '~'+predicate)

    S = list(string)

    for i,c in enumerate(string):
        if c == '|':
            S[i] = '&'
        elif c == '&':
            S[i] = '|'

    return ''.join(S)[1:-1] if flag else '('+''.join(S)+')'
```

```
def skolemization(sentence):
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[A-Z]', statement)

    for match in matches[::-1]:
        statement = statement.replace(match, '')

    for s in statements:
        statement = statement.replace(s, s[1:-1])


import re

def fol_to_cnf(fol):

    statement = fol.replace("<=>", "-")

    while '-' in statement:
        i = statement.index('-')
        statement = new_statement

    statement = statement.replace('=>', '-')

    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') + '[' in statement

    while '~A' in statement:
        i = statement.index('~A')
        statement = list(statement)
        statement[i:] = statement[i+2] + "A"
```

expr = (∀ = [A | ∃])

Output:

[¬animals(g) | love(x,y)] ∉ [¬love[x,y] | animals(y)]
[animals(F(z))] — ¬loves(x, G(x)), [loves(A(x), x]

```
def getAttributes(string):
    expr = '
'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+
'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&':
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string

def Skolemization(sentence):
```

```
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('
]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement




import re


def fol_to_cnf(fol):


    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']&['+ statement[i+1:] +
'=>' + statement[:i] + ']'
```

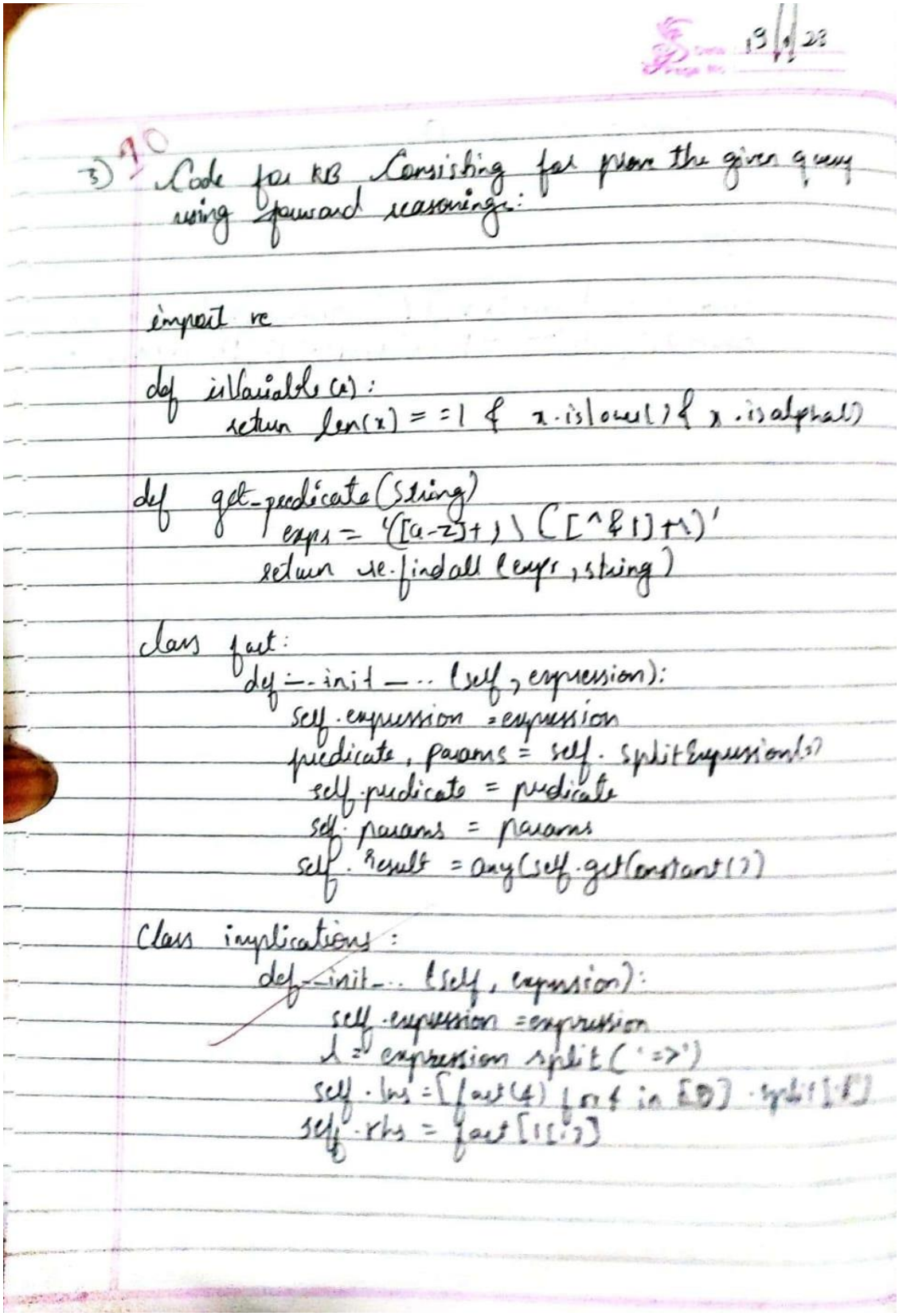
```
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '
'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'
        statement = ''.join(statement)
    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[∀','[~∀')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[∀|∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
```

```
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~
'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement


print(Skolemization(fol_to_cnf("animal(y)<=>loves(x,y)")))
print(Skolemization(fol_to_cnf("∀x[∀y[animal(y)=>loves(x,y)]]=>[∃z[loves(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))
```

OUTPUT

```
Imran-1BM21CS077

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

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



3) Code for KB consisting of FOL to prove the given query using forward reasoning:

```
import re

def isVariable(x):
    return len(x) == 1 & x.islower() & x.isalpha()

def getpredicate(string)
    expr = '([a-z]+)\([^&|]+\)'
    return re.findall(expr, string)

class fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression()
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstant())

Class implications:
    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, fact):
    constants = {}
    new_lhs = []

    for key in constants:
        if constants[key]:
            attributes = attributes.pipline

    return fact(args) if len(new_line) & all
    ([t.getResult() for t in new])
    else none.

class KB:

    def __init__(self):
        self.fact = set()
        self.fact = set()

    def tell(self, e):
        if '=>' in e:
            self.implication.add(implications[e)]
        else:
            self.facts.add[facts(e)]

        for i in self.implications:
            res = i.evaluate(self.facts)
        if res:
            self.facts.add(rs)
```

```
def display(self):
    print("All facts:")
    for i, f in enumerate(set([f.expression
                for f in self.facts])):
        print(f'{i+1}: {f}')

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:

All facts:

Query: evil(x)

1. evil(John)

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = '
'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = '([a-z~]+)[^&|]+
'
    return re.findall(expr, string)


class Fact:
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]
```

```
    def getResult(self):
        return self.result

    def getConstants(self):
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"
        return Fact(f)

class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
```

```
                new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None


class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()


    def tell(self, e):
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)


    def query(self, e):
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
```

```
            i += 1

    def display(self):
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')


kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono,America)')
kb.tell('owns(Nono,M1)')
kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

OUTPUT

```
Imran-1BM21CS077

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