VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
On

ARTIFICIAL INTELLIGENCE
s

Submitted by

SANNIDHI M (1BM21CS189)

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 **SANNIDHI M (1BM21CS189)**, 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.

Madhavi R P
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

INDEX

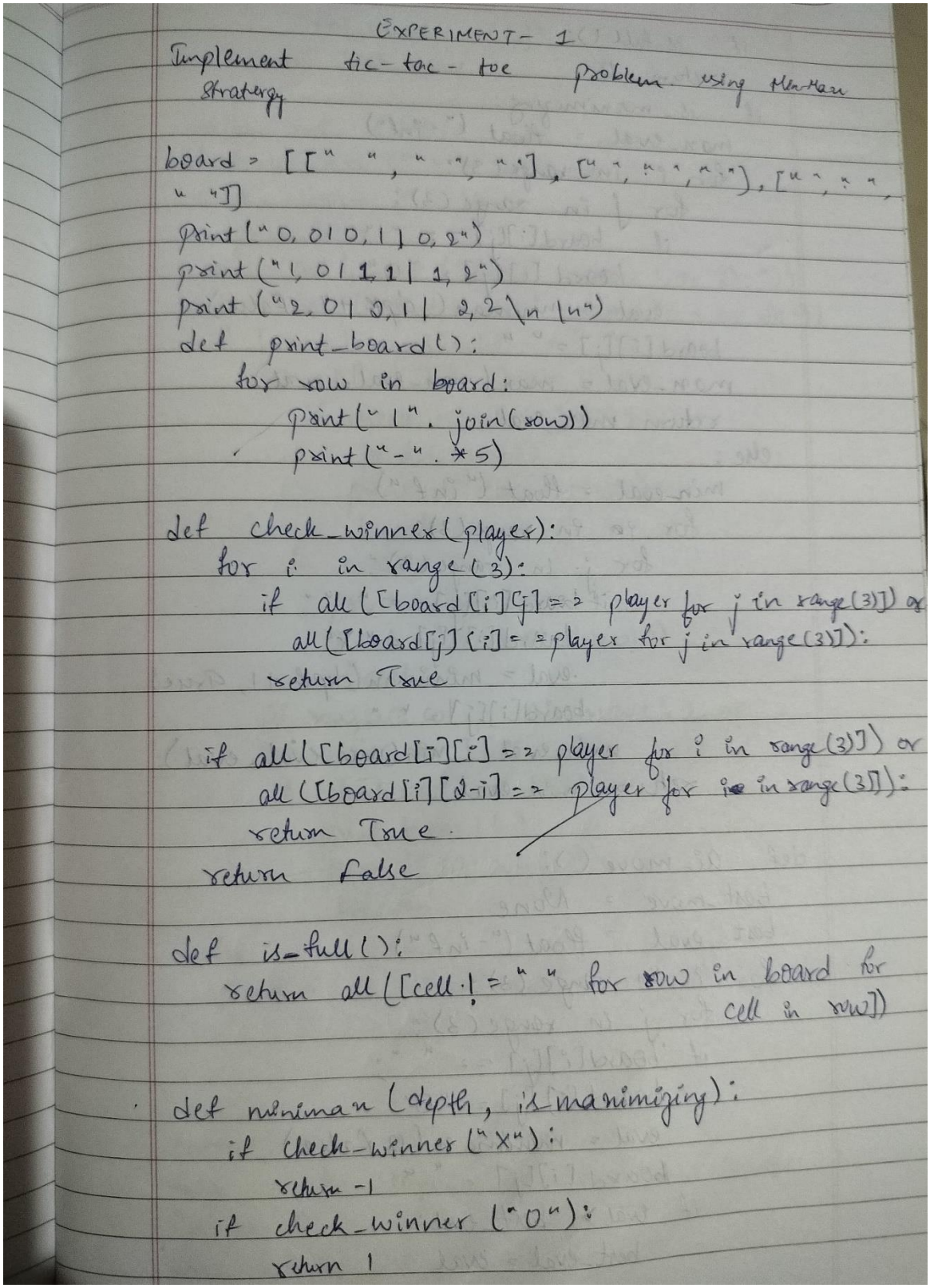
SL No	Name of Experiment	Page No
1	Implement Tic –Tac –Toe Game	1
2	Implement 8 puzzle problem	7
3	Implement Iterative deepening search algorithm.	12
4	Implement A* search algorithm.	16
5	Implement vaccum cleaner agent.	24
6	Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .	31
7	Create a knowledge base using prepositional logic and prove the given query using resolution	34
8	Implement unification in first order logic	40
9	Convert a given first order logic statement into Conjunctive Normal Form (CNF).	45
10	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	52

1.Implement Tic –Tac –Toe Game.

```
board = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]
print("0,0|0,1|0,2")
print("1,0|1,1|1,2")
print("2,0|,2,1|2,2 \n\n")
def print_board():
  for row in board:
    print("|".join(row))
    print("-" * 5)

def check_winner(player):
  for i in range(3):
    if all([board[i][j] == player for j in range(3)]) or all([board[j][i] == player for j in range(3)]):
      return True

  if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):
    return True
  return False

def is_full():
  return all([cell != " " for row in board for cell in row])

def minimax(depth, is_maximizing):
  if check_winner("X"):
    return -1
  if check_winner("O"):
    return 1
  if is_full():
    return 0
  if is_maximizing:
    max_eval = float("-inf")
    for i in range(3):
      for j in range(3):
        if board[i][j] == " ":
          board[i][j] = "O"
          eval = minimax(depth + 1, False)
          board[i][j] = " "
          max_eval = max(max_eval, eval)
    return max_eval
  else:
    min_eval = float("inf")
    for i in range(3):
      for j in range(3):
        if board[i][j] == " ":
            board[i][j] = "X"
            eval = minimax(depth + 1, True)
            board[i][j] = " "
            min_eval = min(min_eval, eval)

    return min_eval

def ai_move():
  best_move = None
```

```
  best_eval = float("-inf")
  for i in range(3):
    for j in range(3):
      if board[i][j] == " ":
        board[i][j] = "O"
        eval = minimax(0, False)
        board[i][j] = " "
        if eval > best_eval:
          best_eval = eval
          best_move = (i, j)

  return best_move

while not is_full() and not check_winner("X") and not check_winner("O"):
  print_board()
  row = int(input("Enter row (0, 1, or 2): "))
  col = int(input("Enter column (0, 1, or 2): "))
  if board[row][col] == " ":
    board[row][col] = "X"
    if check_winner("X"):
      print_board()
      print("You win!")
      break
    if is_full():
      print_board()
      print("It's a draw!")
      break
    ai_row, ai_col = ai_move()
    board[ai_row][ai_col] = "O"
    if check_winner("O"):
      print_board()
      print("AI wins!")
      break

  else:
    print("Cell is already occupied. Try again.")
```

OBSERVATION



EXPERIMENT - 1

Implement tic-tac-toe problem using MinMax Strategy

```
board = [[" ", " ", " "], [" ", " ", " "], [" ", " ", " "]]
print("0,0|0,1|0,2")
print("1,0|1,1|1,2")
print("2,0|2,1|2,2\n|n|n")
def print_board():
    for row in board:
        print("|".join(row))
        print("-"*5)

def check_winner(player):
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or
            all([board[j][i] == player for j in range(3)]):
            return True

    if all([board[i][i] == player for i in range(3)]) or
        all([board[i][2-i] == player for i in range(3)]):
        return True
    return False

def is_full():
    return all([cell != " " for row in board for
                                    cell in row])

def minimax(depth, is_maximizing):
    if check_winner("X"):
        return -1
    if check_winner("O"):
        return 1
```

```
            best_move = (i, j)
    return best_move

while not is_full() and not
check_winner("X") and not
check_winner("O"):
    print_board()
row = int(input("Enter row (0, 1, or 2): "))
col = int(input("Enter column (0, 1, or 2): "))
if board[row][col] == " ":
    board[row][col] = "X"
    if check_winner("X"):
        print_board()
        print("You win!")
        break
    if is_full():
        print_board()
        print("It's a draw!")
        break
    ai_row, ai_col = ai_move()
    board[ai_row][ai_col] = "O"
    if check_winner("O"):
        print_board()
        print("AI wins!")
        break
else:
    print("Cell is already occupied. Try again")
```

OUTPUT

```
[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
```

```
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
+-------------------------------+
```

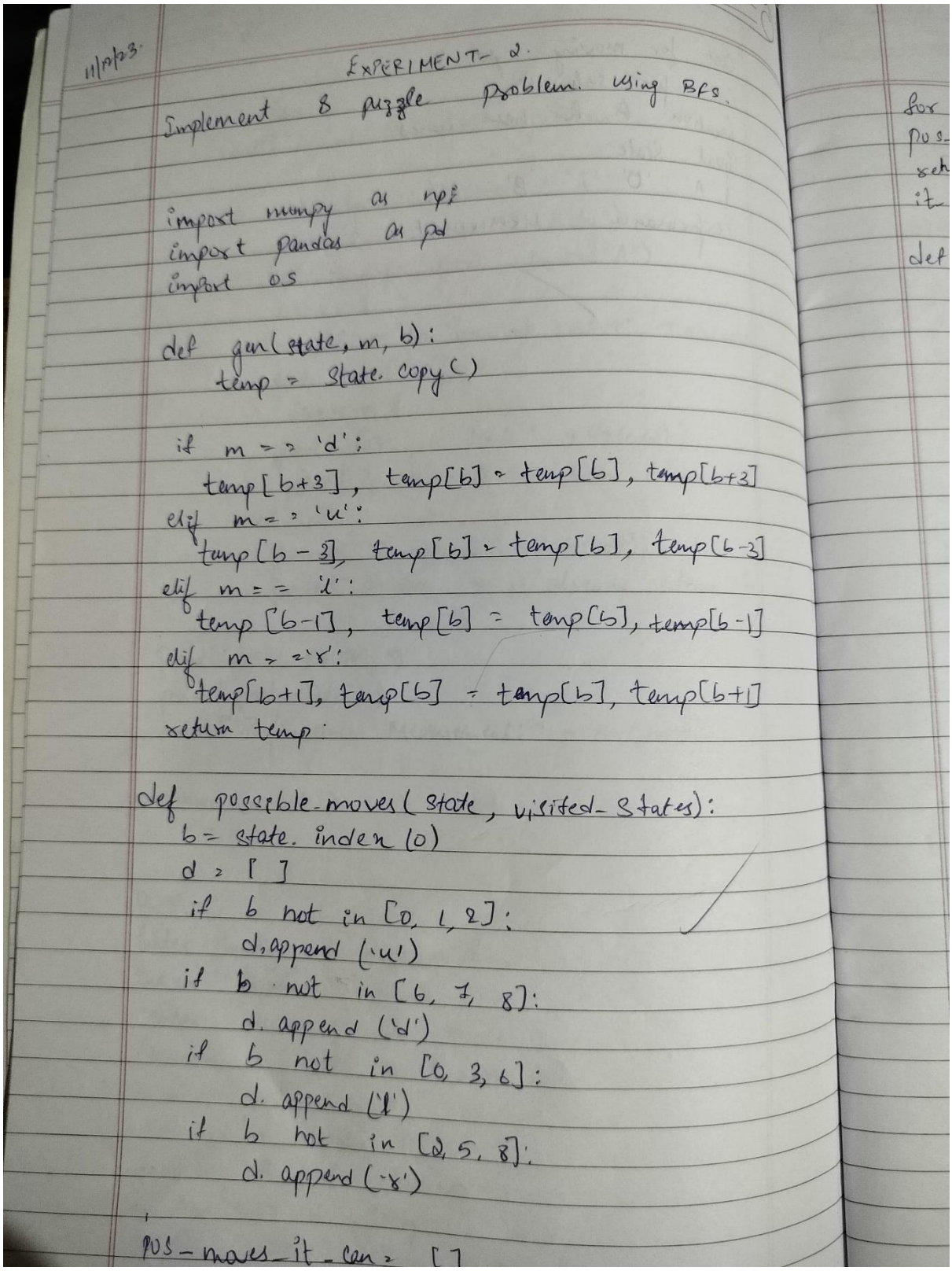
```
+-----------------------------+
Your turn :
enter a number on the board :5
+-----------------------------+
|             |             |             |
|      X      |      X      |      3      |
|             |             |             |
+-----------------------------+
|             |          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

2 .Solve 8 puzzle problems.

```
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)
```

OBSERVATION



11/09/23.

EXPERIMENT- 2.

Implement 8 puzzle problem using BFS.

```
import numpy as np
import pandas as pd
import os

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 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 bfs(src, target):
    queue = [ ]
    queue.append (src)
    exp = [ ]
    while len(queue) > 0:
        source = queue.pop (0)
        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
    pos_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)

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

OUTPUT

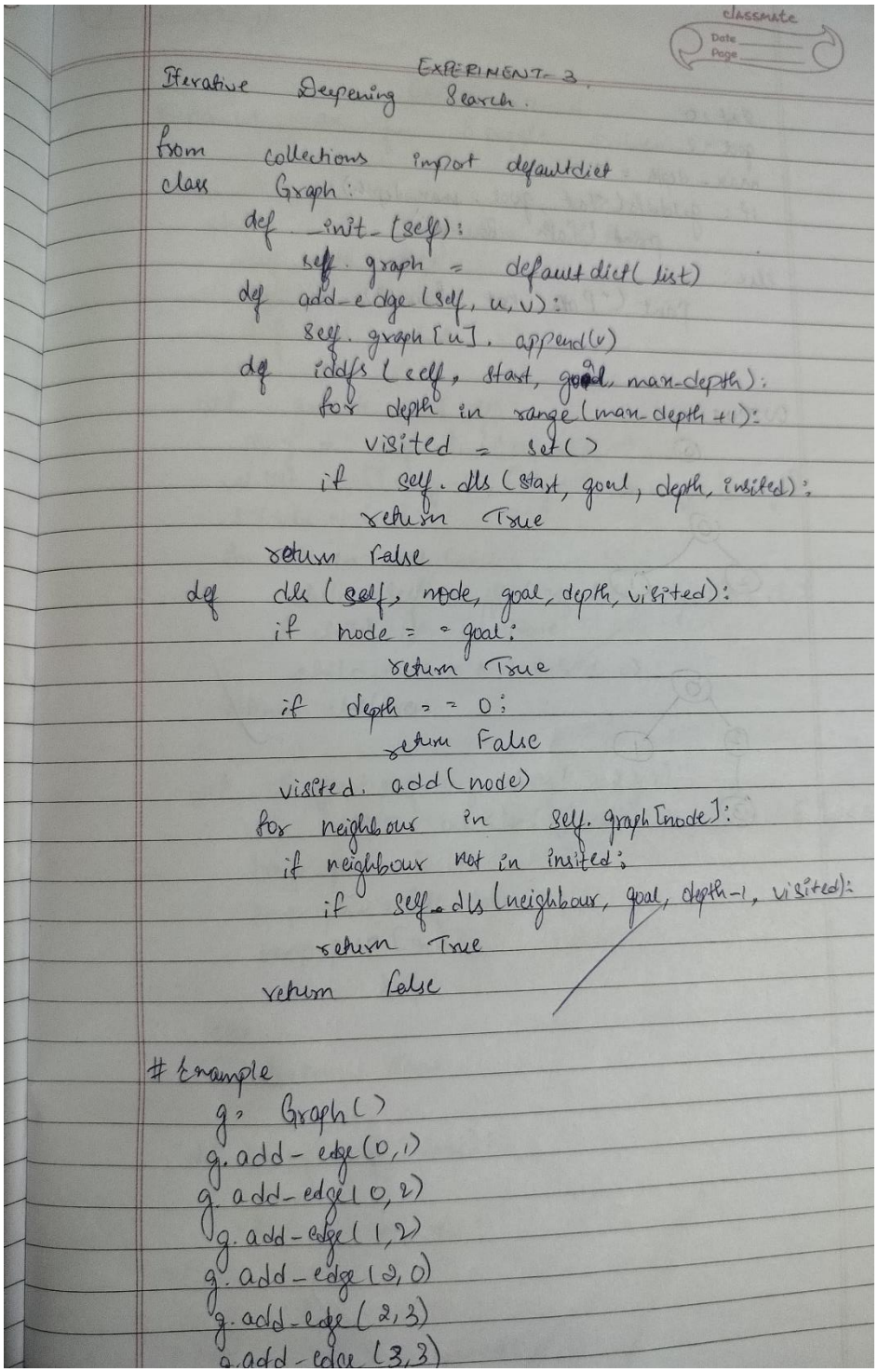
```
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8
-----------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
-----------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8
-----------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8
-----------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8
-----------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0
-----------
Success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

3. Implement Iterative deepening search algorithm.

```
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")
```

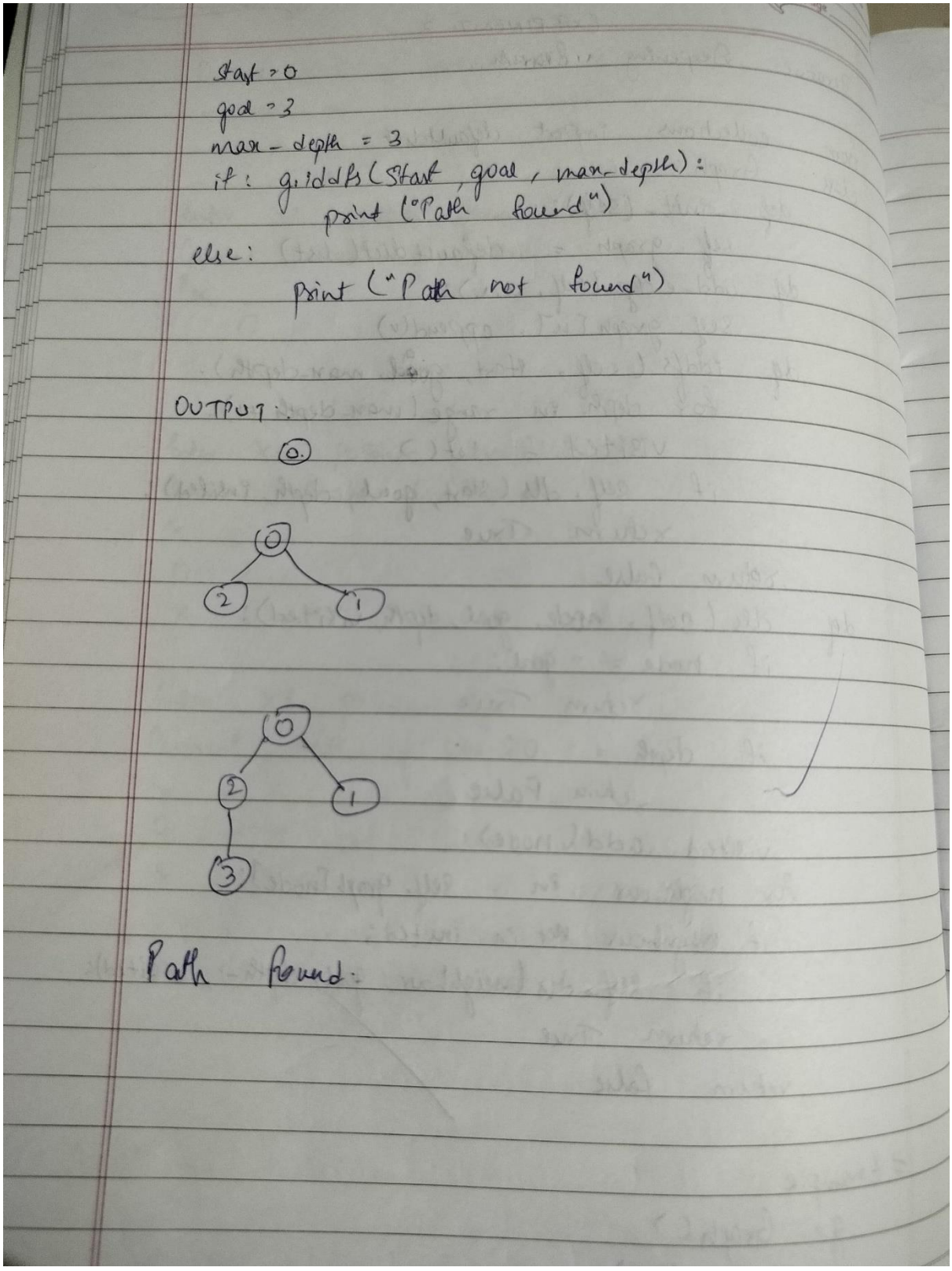
OBSERVATION



EXPERIMENT-3

Iterative Deepening Search.

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def add_edge(self, u, v):
        self.graph[u].append(v)
    def iddfs(self, start, goal, max_depth):
        for depth in range(max_depth+1):
            visited = set()
            if self.dls(start, goal, depth, visited):
                return True
        return False
    def dls(self, node, goal, depth, visited):
        if node == goal:
            return True
        if depth == 0:
            return False
        visited.add(node)
        for neighbour in self.graph[node]:
            if neighbour not in visited:
                if self.dls(neighbour, goal, depth-1, visited):
                    return True
        return False

# Example
g = Graph()
g.add_edge(0,1)
g.add_edge(0,2)
g.add_edge(1,2)
g.add_edge(2,0)
g.add_edge(2,3)
g.add_edge(3,3)
```



OUTPUT

```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [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]]
```

4. Implement A* search algorithm.

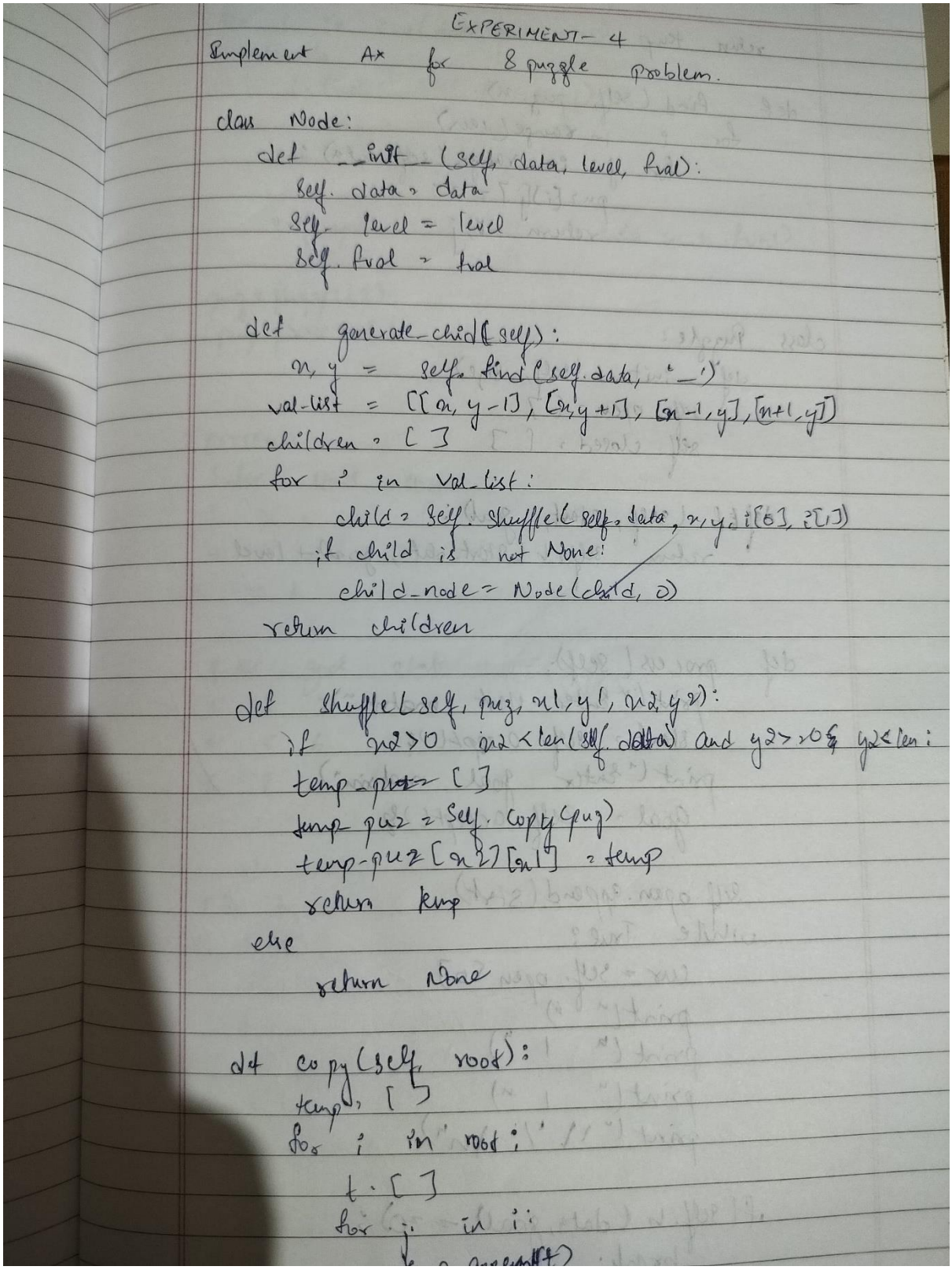
```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    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
```

OBSERVATION

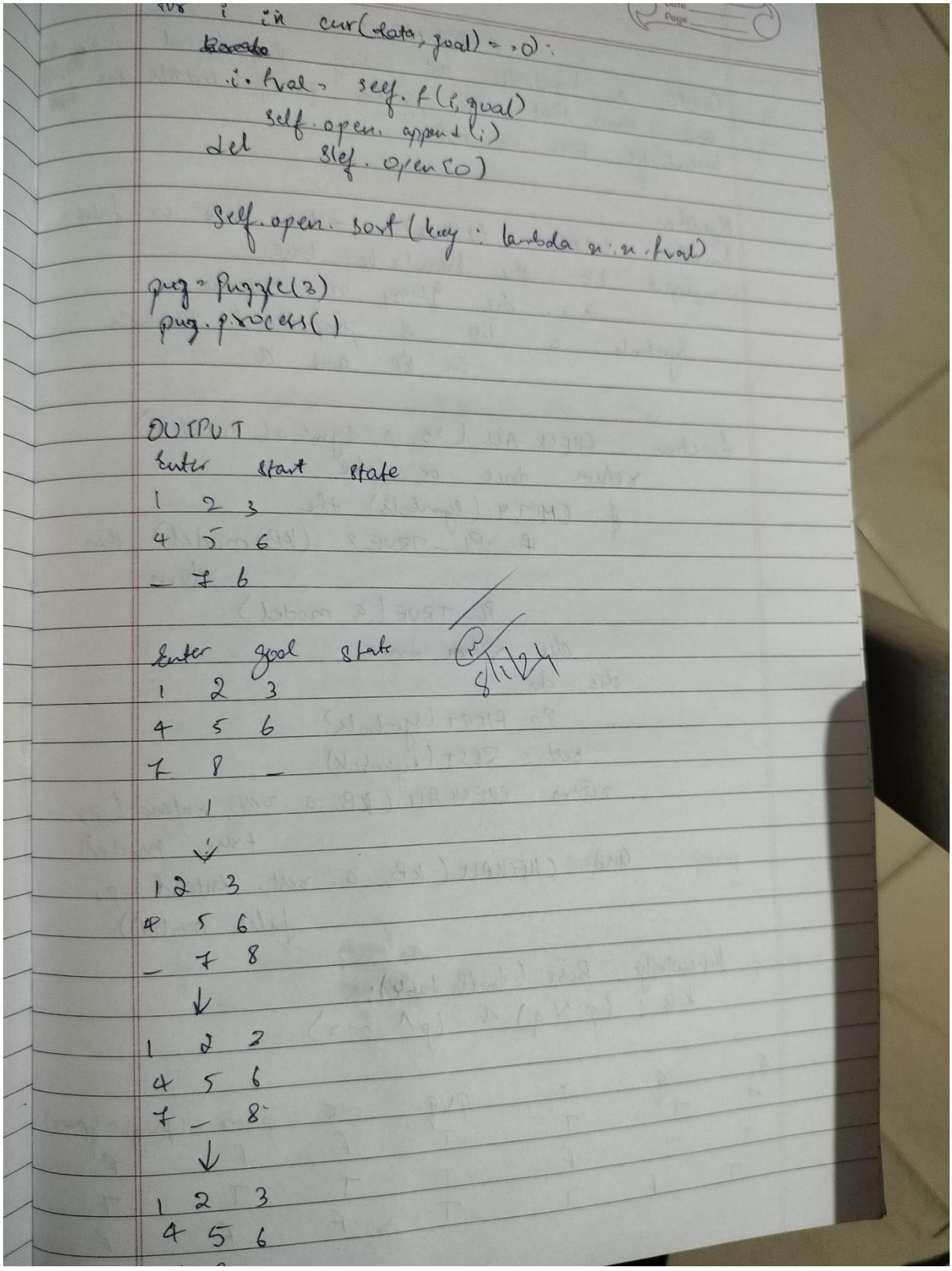


EXPERIMENT- 4

Implement A* for 8 puzzle problem.

```
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, 0)
        return children

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

    def copy(self, root):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
```

```
return temp

def find(self, puz, n):
    for i in range(len)
        for j in range(self.data):
            puz[i][j] = n:
            return i, j


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

    def f(self, start, goal):
        return self.h(start.data, goal) + level


def process(self):
    print("\nEnter start matrix")
    start = self.accept()
    print("\nEnter goal matrix")
    goal = self.accept()

    self.open.append(start)
    while True:
        cur = self.open[0]
        print("  ")
        print("  |  :")
        print("  |  ")
        print(" \|/ :|\n")

    if(self.h(data, goal) == 0):
        break
```



```
for i in cur(data, goal) == 0):
    i.fval = self.f(i, goal)
    self.open.append(i)
del  self.open[0]

self.open.sort(key = lambda x: x.fval)

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

OUTPUT

Enter start state

1 2 3
4 5 6
_ 7 8

Enter goal state

1 2 3
4 5 6
7 8 _

1

↓

1 2 3
4 5 6
_ 7 8

↓

1 2 3
4 5 6
7 _ 8

↓

1 2 3
4 5 6

OUTPUT

```
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 _
```

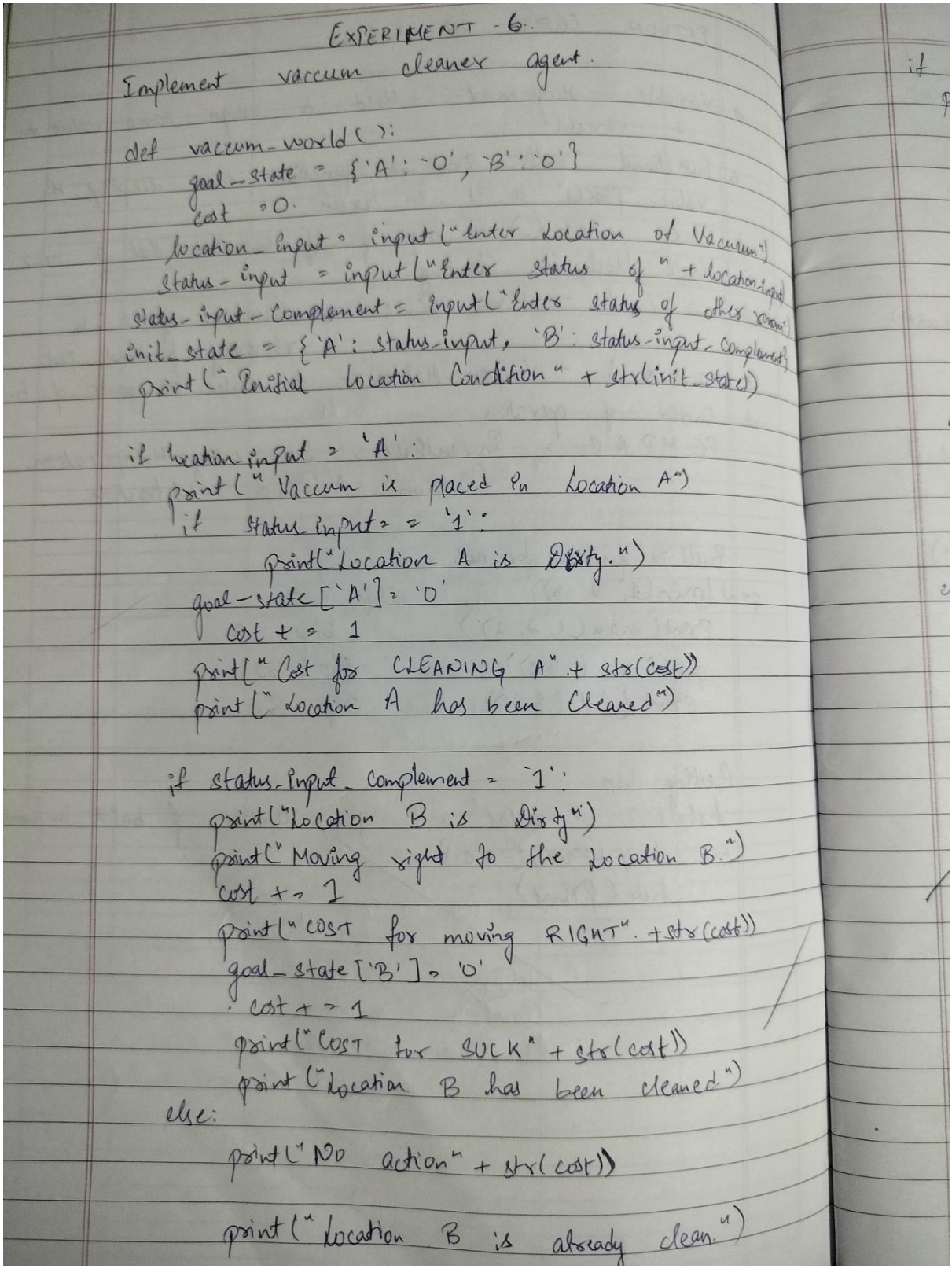
5. Implement vaccum cleaner agent.

```
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))
```

```
            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()
```

OBSERVATION



EXPERIMENT - 6.

Implement vaccum cleaner agent.

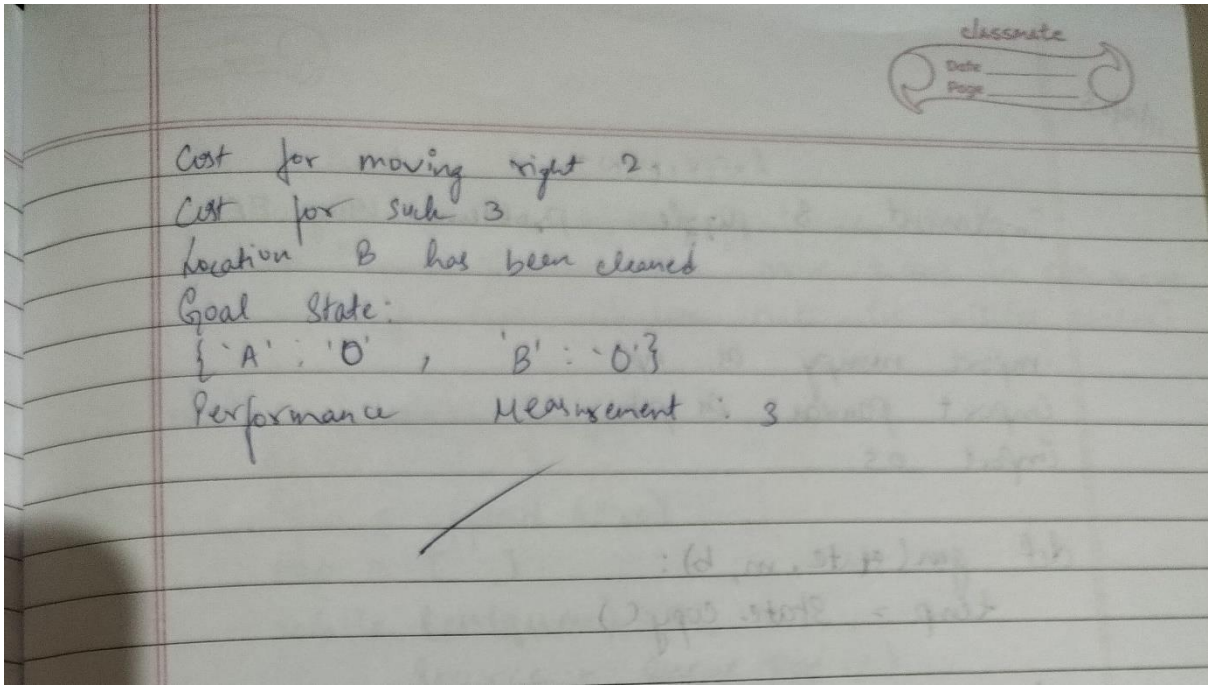
```
def vaccum_world():
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter location of Vacuum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")
    init_state = {'A': status_input, 'B': status_input_complement}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

if location_input == 'A':
    print("Vaccum is placed in location A")
    if status_input == '1':
        print("Location A is Dirty.")
    goal_state['A'] = '0'
    cost += 1
    print("Cost for CLEANING A" + str(cost))
    print("Location A has been Cleaned")

if status_input_complement == '1':
    print("Location B is Dirty")
    print("Moving right to the location B.")
    cost += 1
    print("COST for moving RIGHT" + str(cost))
    goal_state['B'] = '0'
    cost += 1
    print("COST for SUCK" + str(cost))
    print("Location B has been cleaned")
else:
    print("No action" + str(cost))

    print("Location B is already clean.")
```

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

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

    if status_input_complement == '1':
        print("Location A is Dirty.")
        print("Moving LEFT to the location A")
        cost += 1.
        print("COST for moving LEFT" + str(cost))
        goal_state['A'] = 0
        cost += 1.
        print("COST for SUCK", + str(cost))
        Cleaned.")
```

```
else:
    print(cost)
    print("Location B is already clean")

if status_input_complement == '1':
    print("Location A is Dirty")
    print("Moving left to 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 Status of A 1
Enter status of other room 1
Vacuum is placed in location A
Location is dirty
Cost for cleaning A 1
Location A is cleaned
Location B is dirty
Moving right to the location B
```



OUTPUT:

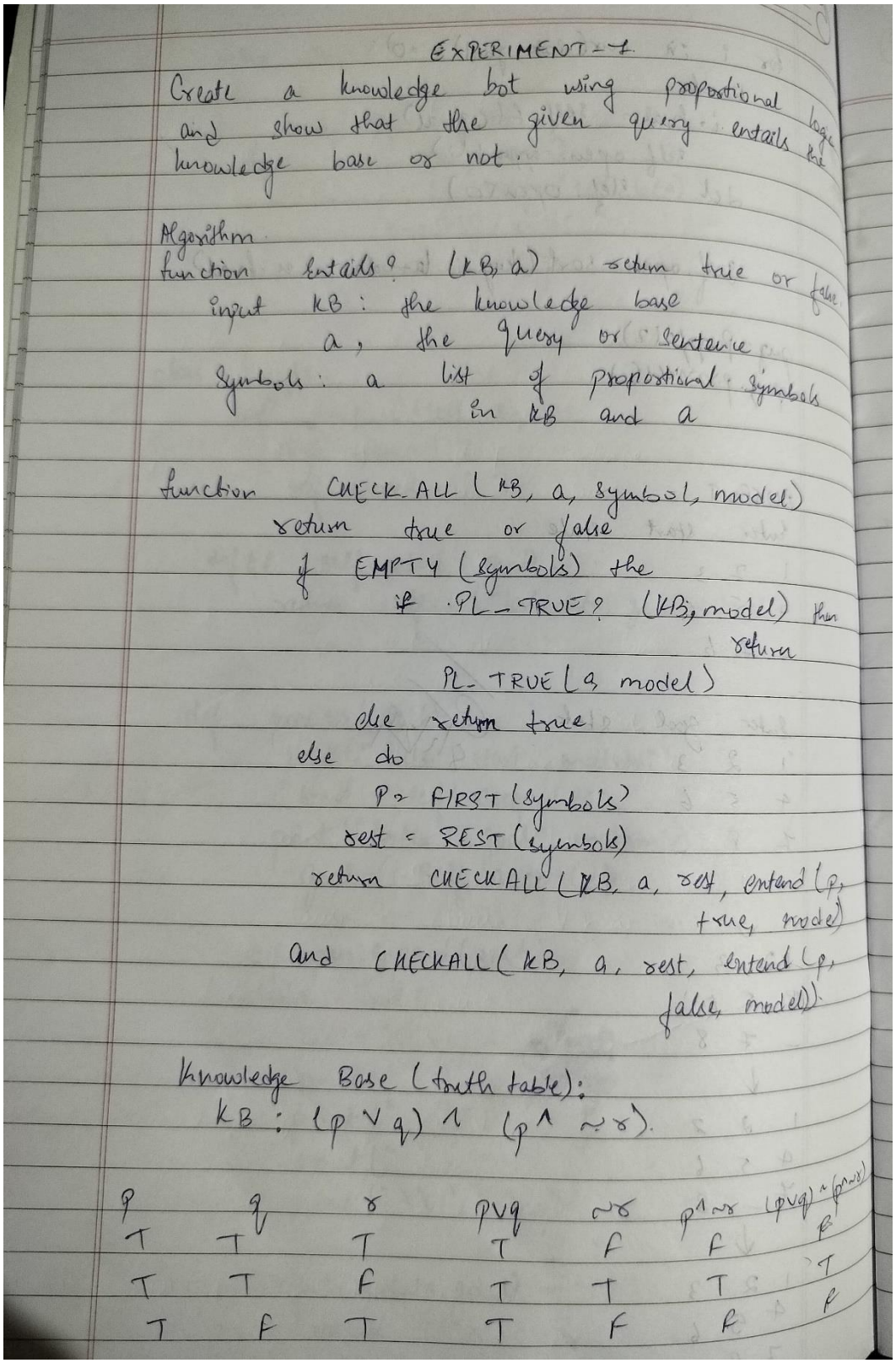
```
0 indicates clean and 1 indicates dirty
Enter Location of Vacuumb
Enter status of b1
Enter status of other room1
Vacuum is placed in location B
Location B is Dirty.
COST for CLEANING 1
Location B has been Cleaned.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT2
COST for 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 .

```
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)))
    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)

OBSERVATION



EXPERIMENT-7

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

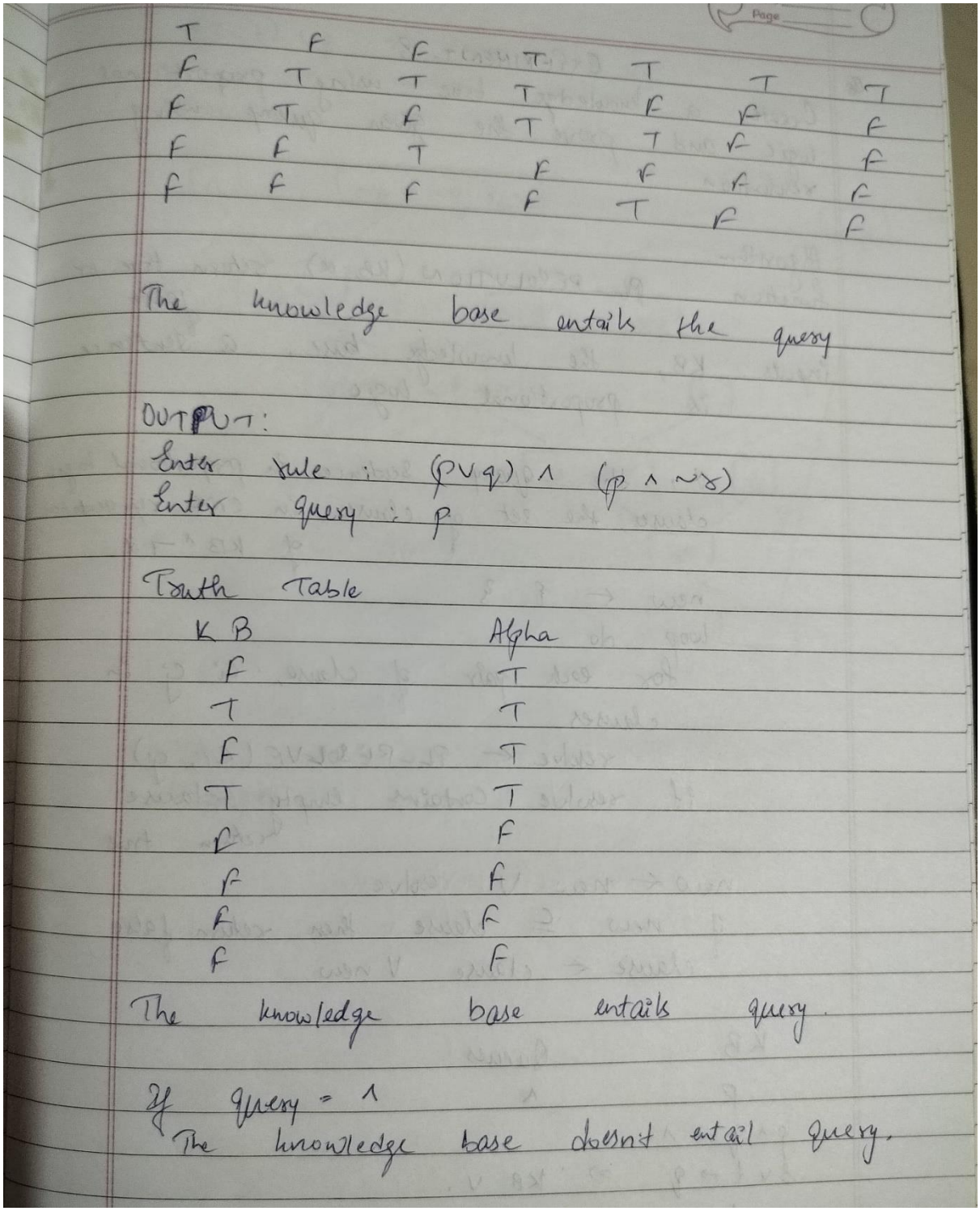
Algorithm

function entails? (KB, a) return true or false
  input KB: the knowledge base
    a, the query or sentence
  Symbols: a list of proposition symbols in KB and a

function CHECK_ALL (KB, a, symbol, model)
  return true or false
  if EMPTY (symbols) then
    if PL-TRUE? (KB, model) then return PL_TRUE (a, model)
    else return true
  else do
    P = FIRST (symbols)
    rest = REST (symbols)
    return CHECKALL (KB, a, rest, extend (p, true, model)
    and CHECKALL (KB, a, rest, extend (p, false, model)).

Knowledge Base (truth table):
KB: (p ∨ q) ∧ (p ∧ ~r).

p	q	r	p∨q	~r	p∧~r	(p∨q)∧(p∧~r)
T	T	T	T	F	F	F
T	T	F	T	T	T	T
T	F	T	T	F	F	F



T F F T T T T
F T T T F F F
F T F T T F F
F F T F F F F
F F F F T F F

The knowledge base entails the query

OUTPUT:
Enter rule : $(p \vee q) \wedge (p \wedge \sim s)$
Enter query : p

Truth Table

KB	Alpha
F	T
T	T
F	T
T	T
F	F
F	F
F	F
F	F

The knowledge base entails query.

If query = 1
The knowledge base doesn't entail query.

OUTPUT:

```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

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')
```

OUTPUT:

```
['~P', 'R']
```

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


def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
```

```
                if gen[0] != negate(gen[1]):
                    clauses += [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.
```

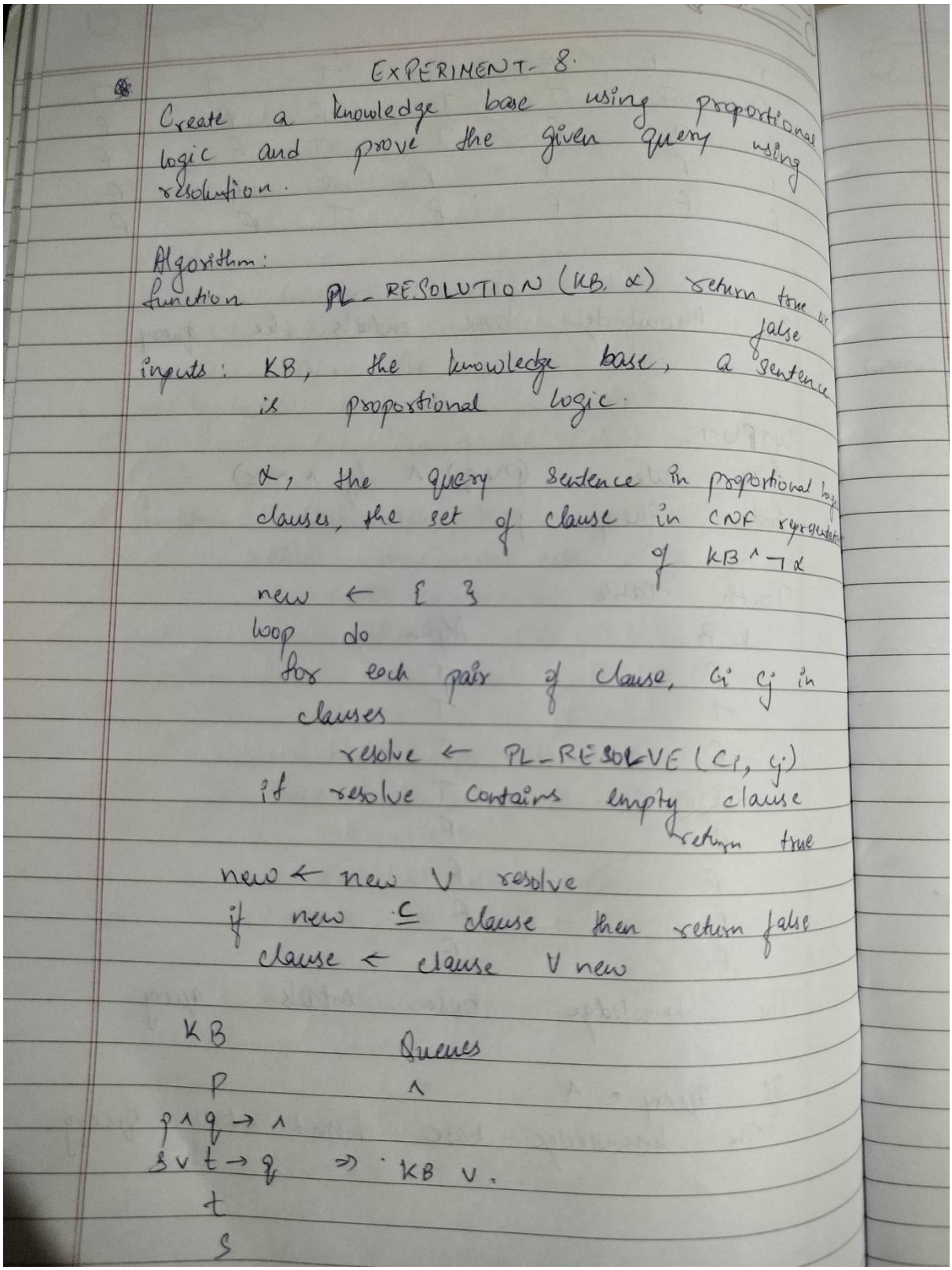
rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR

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.
```

OBSERVATION



EXPERIMENT- 8.

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

Algorithm:

function PL-RESOLUTION (KB, α) return true or false

inputs: KB, the knowledge base, a sentence in proportional logic.

α, the query sentence in proportional logic

clauses, the set of clause in CNF representation of KB ∧ ¬α

new ← { }

loop do

for each pair of clause, Ci Cj in clauses

resolve ← PL-RESOLVE (Ci, Cj)

if resolve contains empty clause return true

new ← new ∪ resolve

if new ⊆ clause then return false

clause ← clause ∪ new

KB

p

p ∧ q → r

s ∨ t → q

t

s

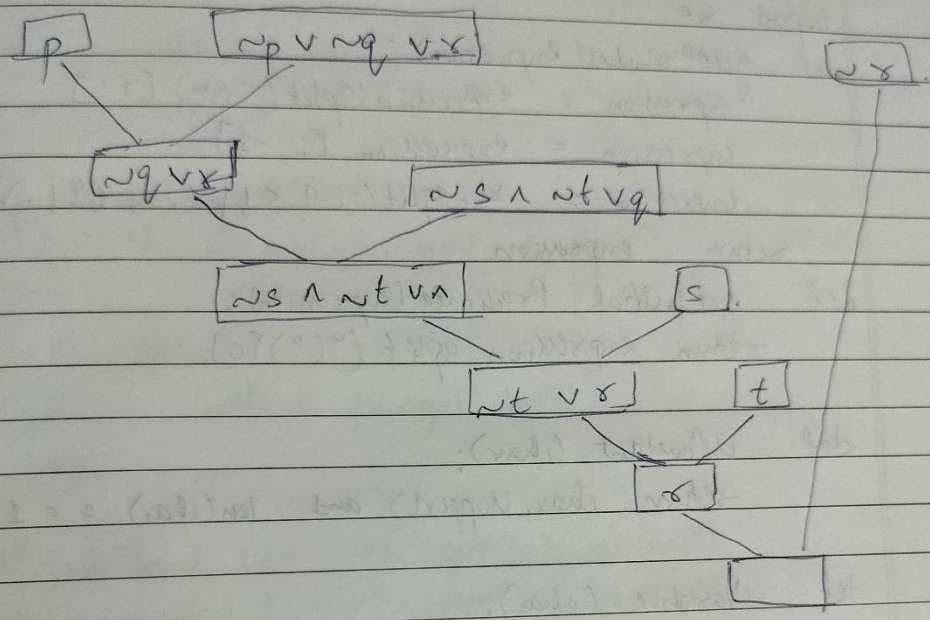
Queries

r

⇒ KB ∪ .

$\sim(s \vee t) \vee q \rightarrow \sim s \wedge \sim t \vee q$

Resolution:



OUTPUT:

true.

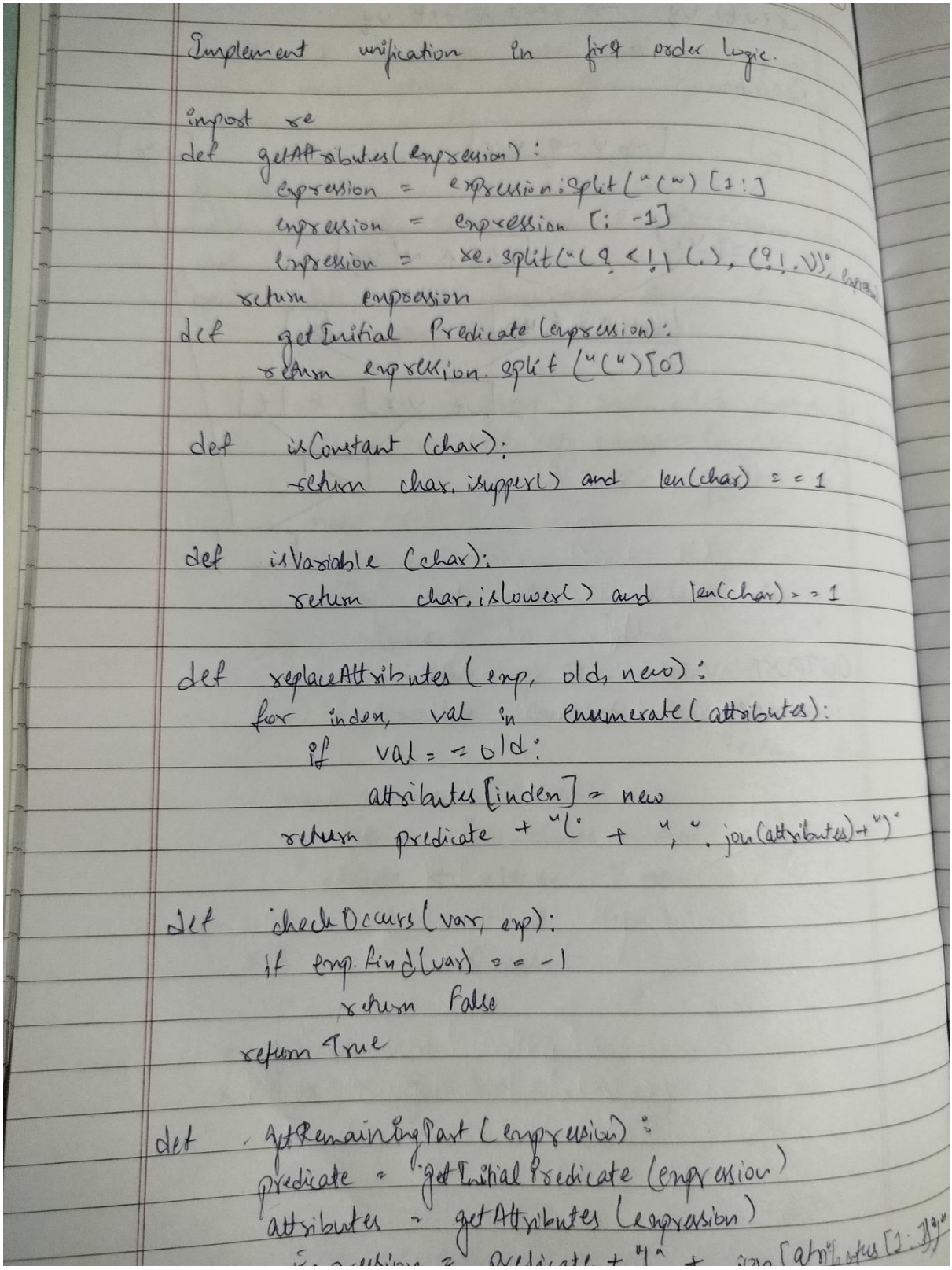
8. Implement unification in first order logic

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

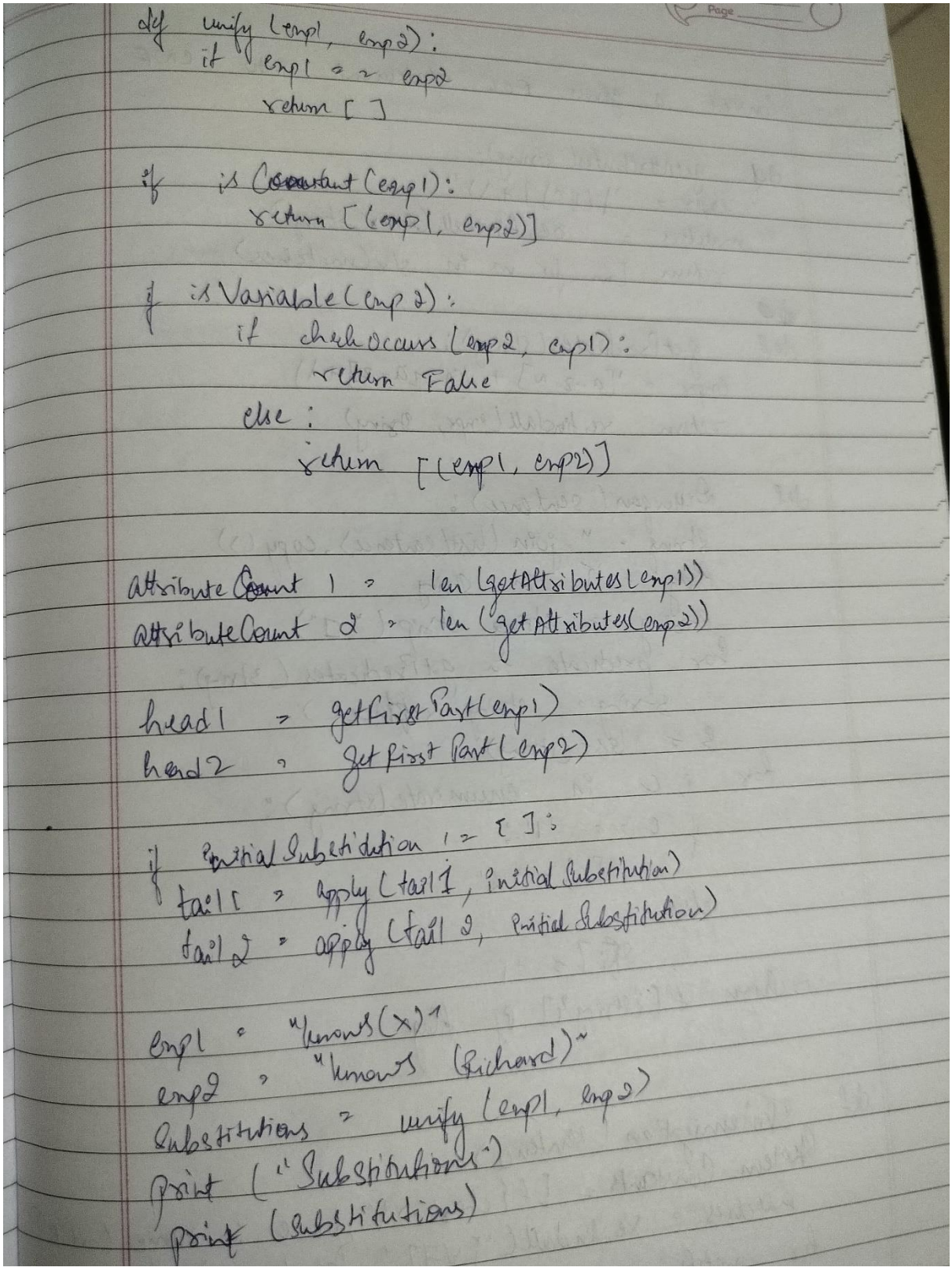
```
    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        return False
    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        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)
```

OBSERVATION



Implement unification in first order logic.

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = expression[:-1]
    expression = re.split("(?<!\(),(?!\))", expression[0])
    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):
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

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

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



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

if   is Constant(exp1):
        return [(exp1, exp2)]

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

attributeCount 1 = len(getAttributes(exp1))
attributeCount 2 = len(getAttributes(exp2))

head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)

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

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

OUTPUT

```
Substitutions:
[('X', 'Richard')]
```

```
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

```
Substitutions:
[('A', 'y'), ('mother(y)', 'x')]
```

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

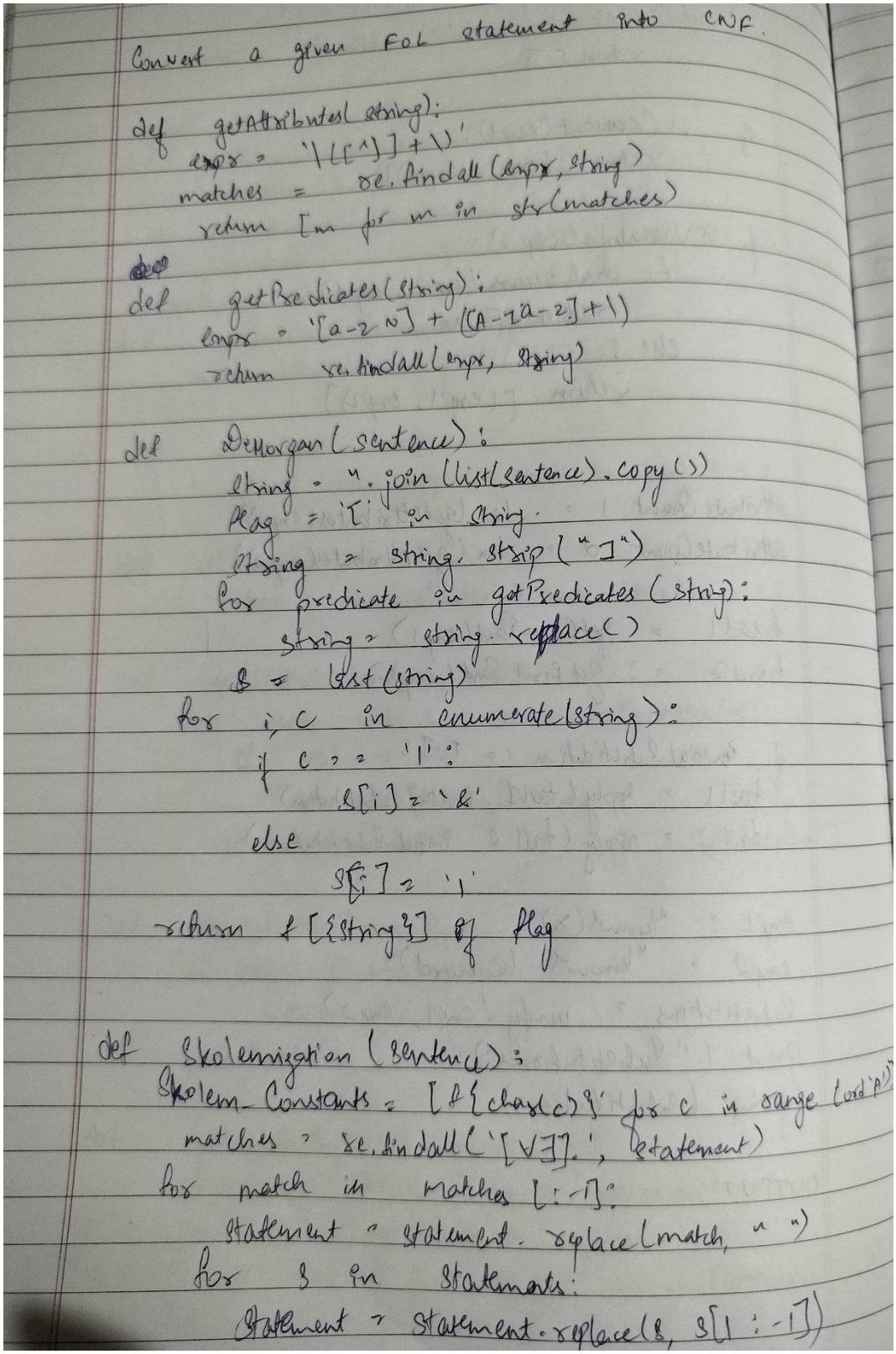
```
def getAttributes(string):
    expr = '
'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = '[a-z~]+
'
    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)"))
```

OBSERVATION

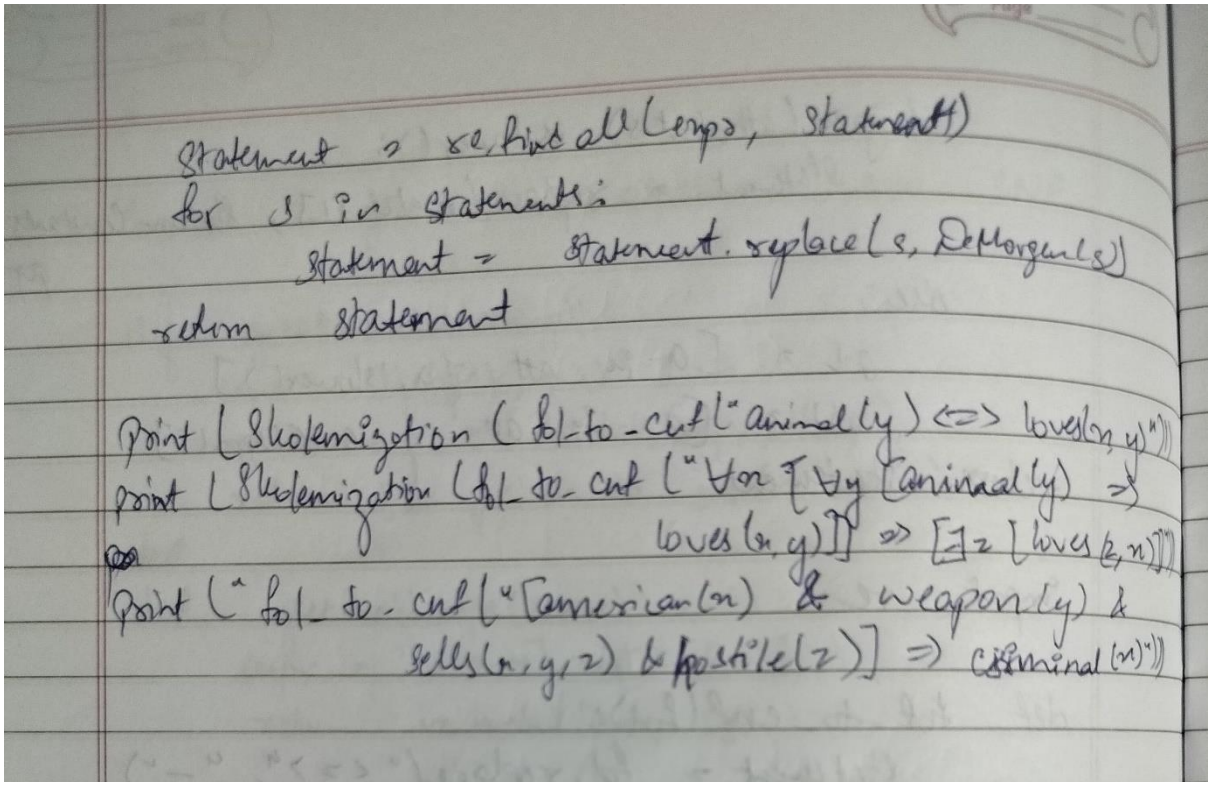


Convert a given FOL statement into CNF.

```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches)
def
def getPredicates(string):
    expr = '[a-zA-Z~]+\([A-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    flag = '[' in string
    string = string.strip("]")
    for predicate in getPredicates(string):
        string = string.replace()
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        else
            s[i] = '|'
    return ''.join(s[1:-1]) if flag
```

```
def Skolemization(sentence):
    Skolem_Constants = [f'{chr(c)}' for c in range(ord('A'))
    matches = re.findall('[∀∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, "")
    for s in statements:
        statement = statement.replace(s, s[1:-1])
```

```
if join(attributes).islower():
    statement = replace(match[1], Student-Courses.pop(0))
else:
    aL = [a in att if a.islower()]
    aU = [a in att if a.isupper()]
return statement

import re

def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement
        i = statement.index("_")
        statement = new_statement
    statements = re.findall(expr)
for i, s in enumerate
    if ']' in s and '[' not in s:
        statement[i] += ']'
for s in statements
    statement = replace(s, fol_to_cnf(s))
    while '-' in statement
        i = statement.index("-")
        br = statement.index("[")
        new_statement = '~' + statement[br:i]
        statement = statement[:5x]
    while '~∀' in statement:
        i = statement.index('~∀')
    statement = list(statement)
    statement = "".join(statement)
while '~∃' in statement:
    p = statement.index('~∃')
    s = list(statement)
    expr = '([~[∀|∃].)'
```



OUTPUT

```
[~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

```
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:
```

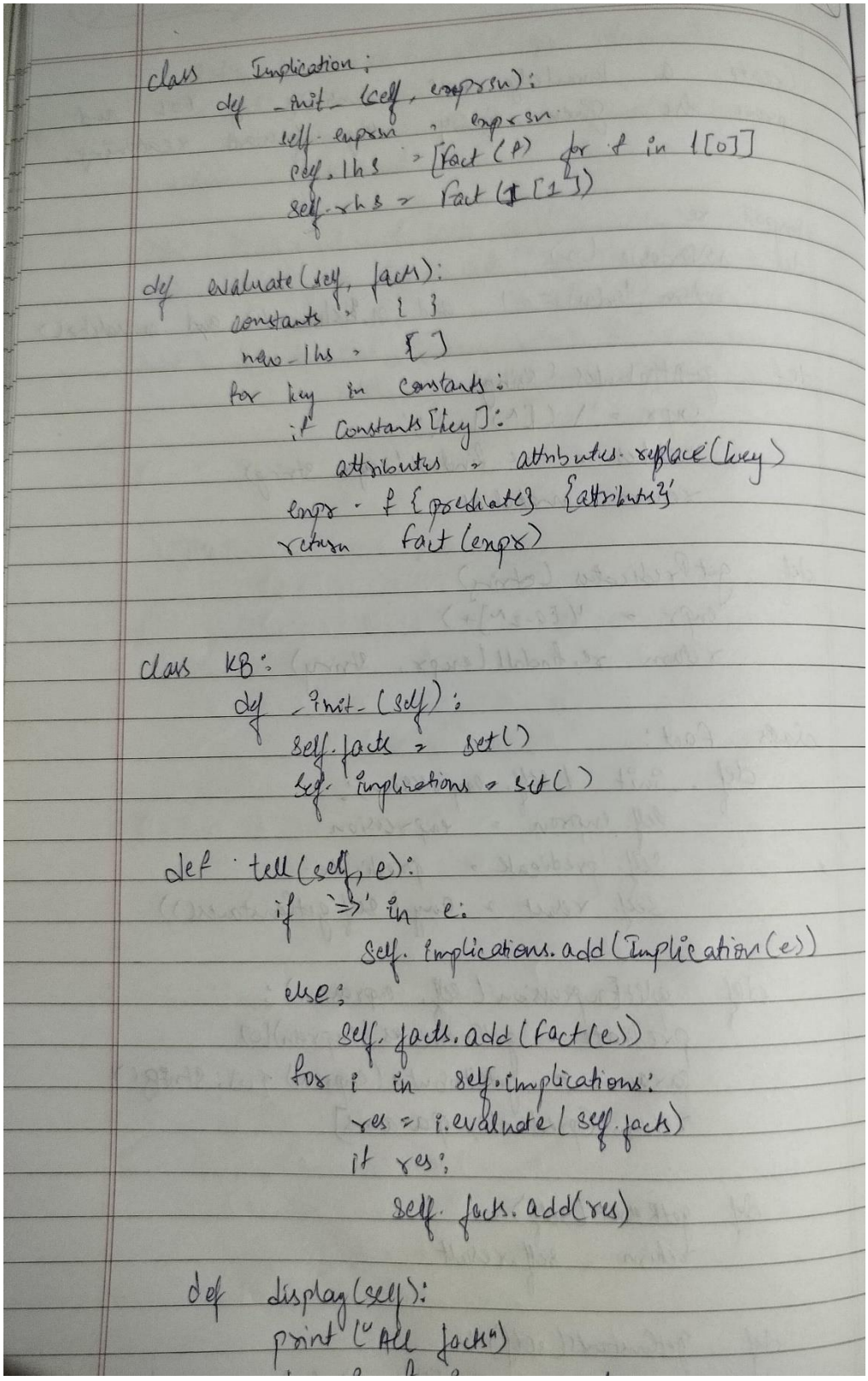
```
                    constants[v] = fact.getConstants()[i]
                new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = 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()
```

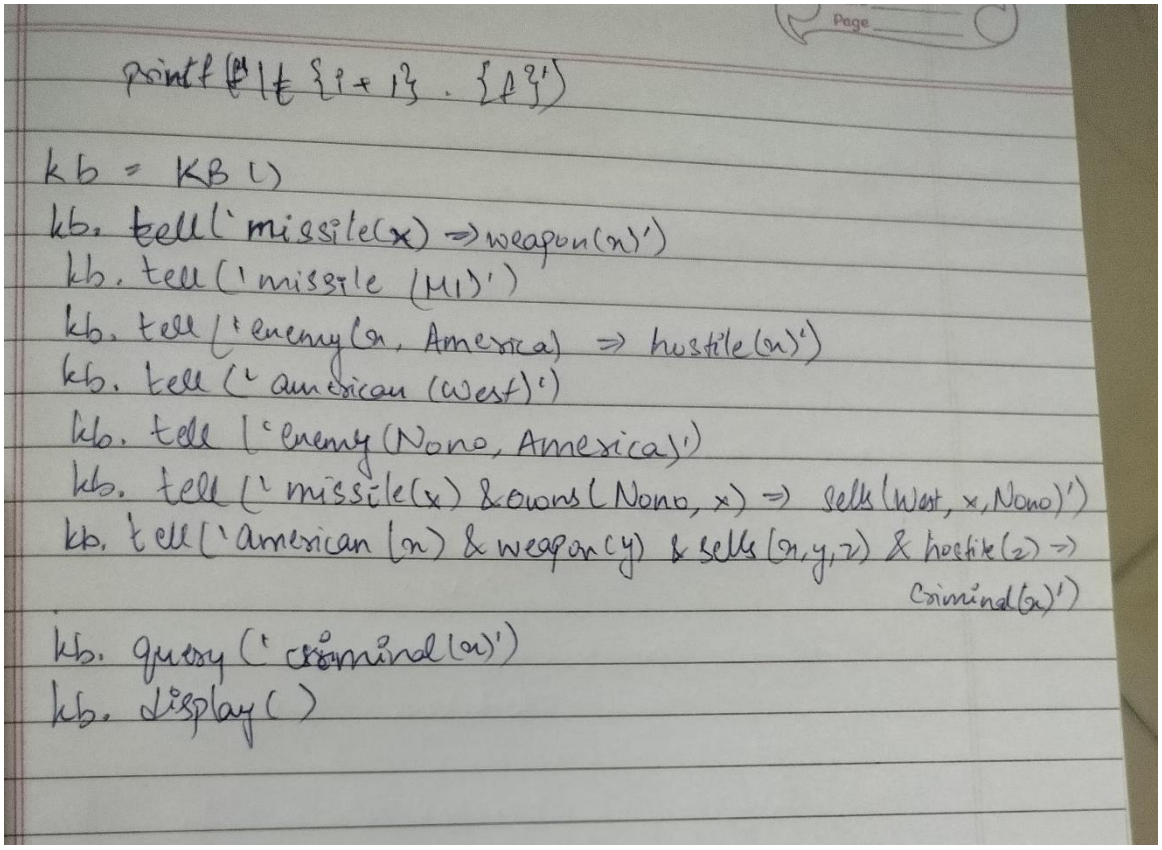
OBSERVATION

Create a knowledge base consisting of FOL 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.expr = expression
        self.predicate = predicate
        self.result = any(self.getConstants())

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

    def getResult(self):
        return self.result

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



```
class Implication:
    def __init__(self, expression):
        self.expr = expression
        self.lhs = [Fact(p) for p in l[0]]
        self.rhs = Fact(l[1])

def evaluate(self, facts):
    constants = {}
    new_lhs = []
    for key in constants:
        if constants[key]:
            attributes = attributes.replace(key)
    expr = f'{predicate}{attributes}'
    return Fact(expr)


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 display(self):
    print("All facts")
```



OUTPUT

```
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)
```