VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence

Submitted by

Anoshor B. Paul (1BM21CS024)

Under the Guidance of

Dr. K. Panimozhi
Assistant Professor, BMSCE

in partial fulfilment 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

November 2023-February 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 **Anoshor B. Paul (1BM21CS024)** , who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfilment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023-24.

The Lab report has been approved as it satisfies the academic requirements in respect of **Artificial Intelligence - (22CS5PCAIN)** work prescribed for the said degree.

Dr. K. Panimozhi	Dr. Jyothi Nayak
Assistant professor	Professor and Head
Department of CSE	Department of CSE
BMSCE, Bengaluru	BMSCE, Bengaluru

B. M. S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

DECLARATION

I, Anoshor B. Paul (1BM21CS024), student of 5th Semester, B.E, Department of Computer Science and Engineering, B. M. S. College of Engineering, Bangalore, here by declare that, this lab report entitled " **Artificial Intelligence** " has been carried out by me under the guidance of Dr. K. Panimozhi, Assistant Professor, Department of CSE, B. M. S. College of Engineering, Bangalore during the academic semester November-2023-February-2024.

I also declare that to the best of my knowledge and belief, the development reported here is not from part of any other report by any other students.

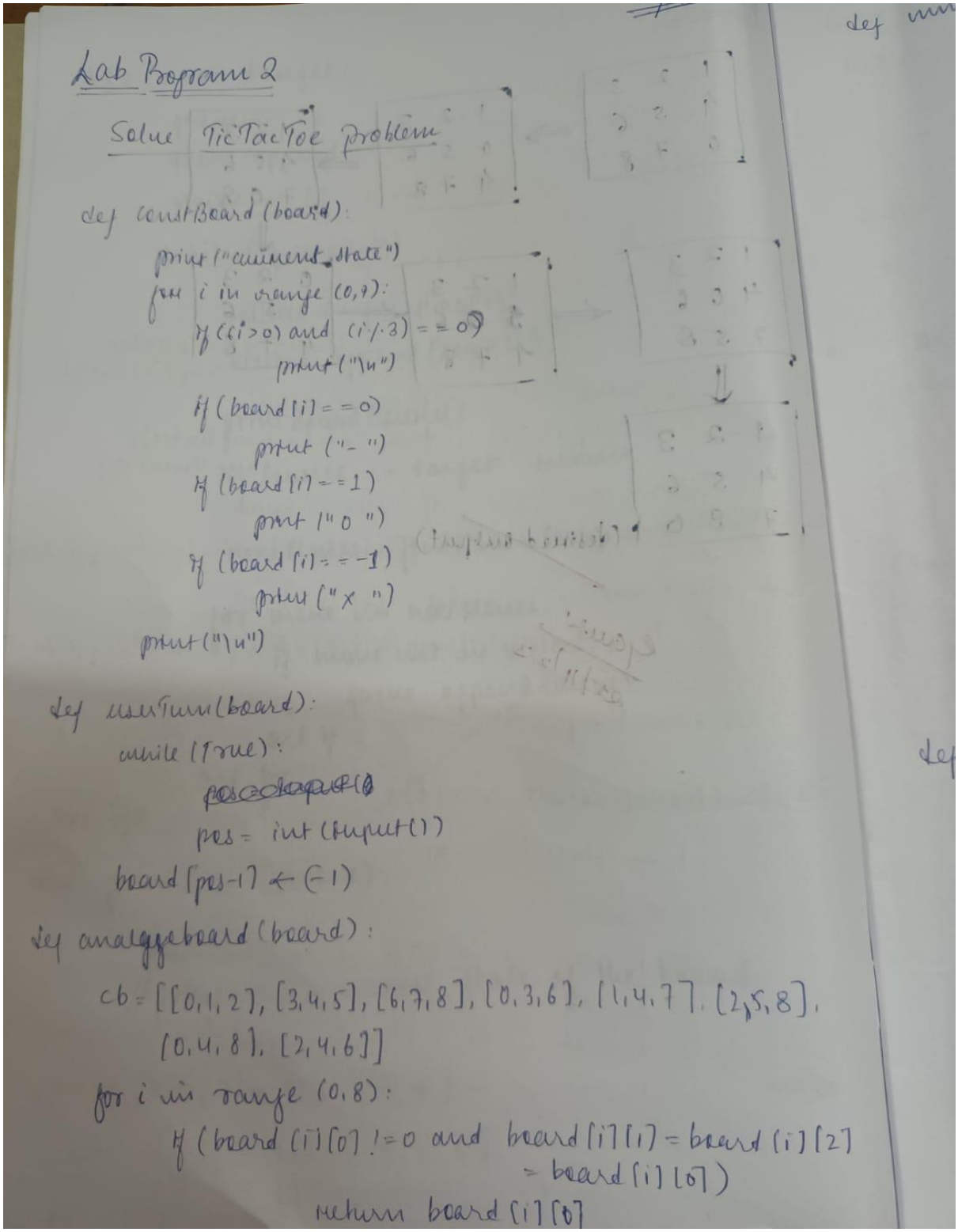
TABLE OF CONTENTS

EXPERIMENT: - 1

Aim

Implement Tic –Tac –Toe Game.

Observation Notebook

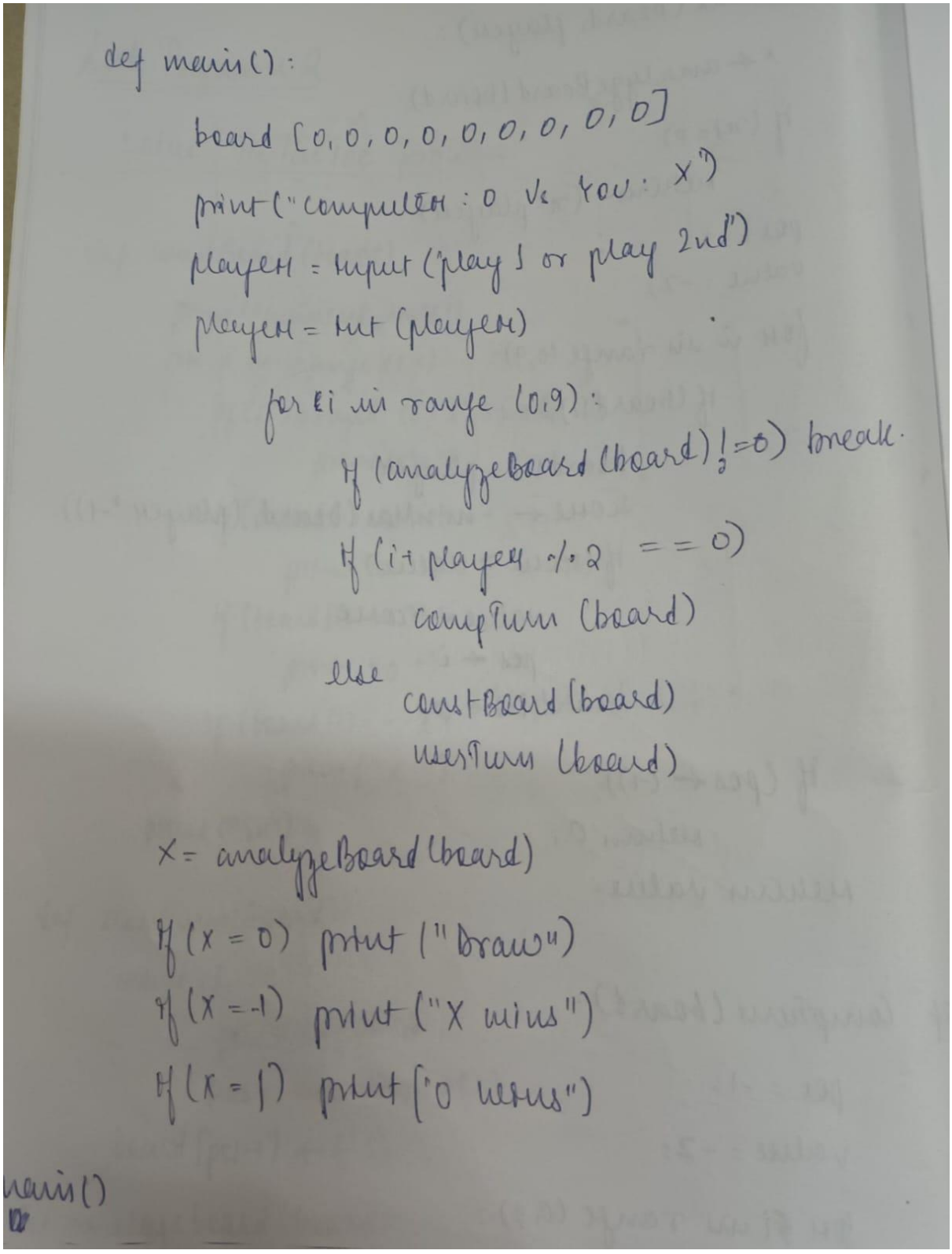


Lab Program 2

Solve TicTacToe problem

```
def constBoard(board):
    print("current state")
    for i in range(0,9):
        if((i>0) and (i%3)==0)
            print("\n")
        if(board[i]==0)
            print("- ")
        if(board[i]==1)
            print("0 ")
        if(board[i]==-1)
            print("x ")
    print("\n")

def userTurn(board):
    while(True):
        pos=int(input())
    board[pos-1]<-(-1)

def analyzeboard(board):
    cb=[[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],
        [0,4,8],[2,4,6]]
    for i in range(0,8):
        if(board[i][0]!=0 and board[i][1]=board[i][2]
                          =board[i][0])
            return board[i][0]
```

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

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

    board[pos] = i;
```



```
def main():
    board [0,0,0,0,0,0,0,0,0]
    print("computer : 0 Vs You : X")
    player = input('play 1 or play 2nd')
    player = int(player)
    for i in range(0,9):
        if(analyzeBoard(board)!=0) break.
        if(i+player %.2 == 0)
            compTurn(board)
        else
            userTurn(board)
            constBoard(board)
    x= analyzeBoard(board)
    if(x=0) print("Draw")
    if(x=-1) print("X wins")
    if(x=1) print('O wins")
main()
```

Code

```
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")
```

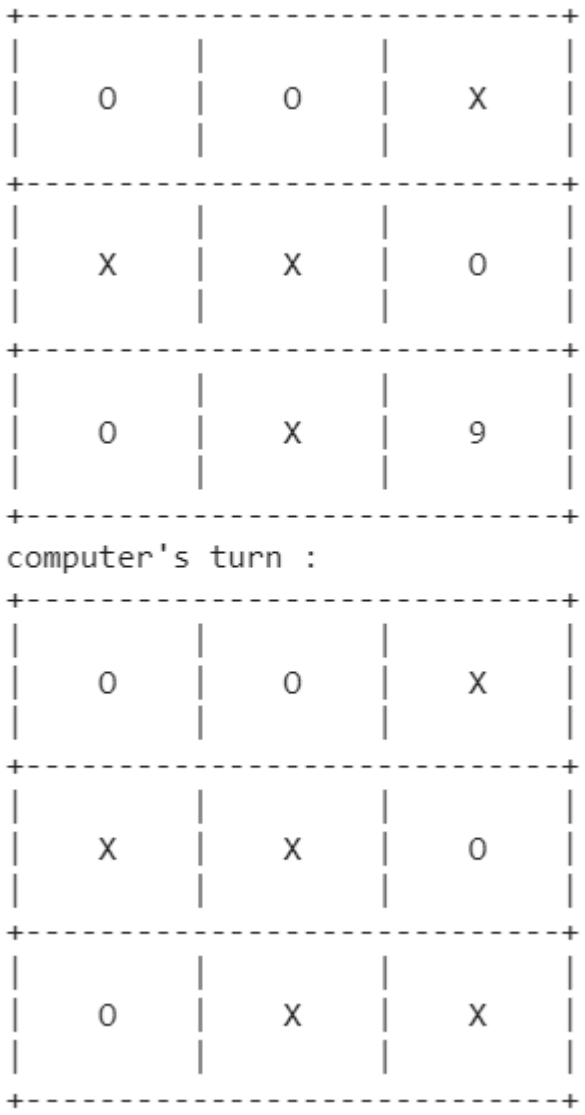
Output

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

```
+-------------------------------+
|               |               |
|       0       |       2       |       3       |
|               |               |
+-------------------------------+
|               |               |
|       4       |       X       |       6       |
|               |               |
+-------------------------------+
|               |               |
|       7       |       8       |       9       |
|               |               |
+-------------------------------+
computer's turn :
+-------------------------------+
|               |               |
|       0       |       2       |       3       |
|               |               |
+-------------------------------+
|               |               |
|       4       |       X       |       6       |
|               |               |
+-------------------------------+
|               |               |
|       7       |       X       |       9       |
|               |               |
+-------------------------------+
Your turn :
enter a number on the board :2
```

```
+-------------------------------+
|         |         |           |
|    0    |    0    |    3      |
|         |         |           |
+-------------------------------+
|         |         |           |
|    4    |    X    |    6      |
|         |         |           |
+-------------------------------+
|         |         |           |
|    7    |    X    |    9      |
|         |         |           |
+-------------------------------+
computer's turn :
+-------------------------------+
|         |         |           |
|    0    |    0    |    X      |
|         |         |           |
+-------------------------------+
|         |         |           |
|    4    |    X    |    6      |
|         |         |           |
+-------------------------------+
|         |         |           |
|    7    |    X    |    9      |
|         |         |           |
+-------------------------------+
Your turn :
enter a number on the board :7
```

```
+-------------------------------+
|               |               |
|       0       |       0       |       X       |
|               |               |
+-------------------------------+
|               |               |
|       4       |       X       |       6       |
|               |               |
+-------------------------------+
|               |               |
|       0       |       X       |       9       |
|               |               |
+-------------------------------+
computer's turn :
+-------------------------------+
|               |               |
|       0       |       0       |       X       |
|               |               |
+-------------------------------+
|               |               |
|       X       |       X       |       6       |
|               |               |
+-------------------------------+
|               |               |
|       0       |       X       |       9       |
|               |               |
+-------------------------------+
Your turn :
enter a number on the board :6
```

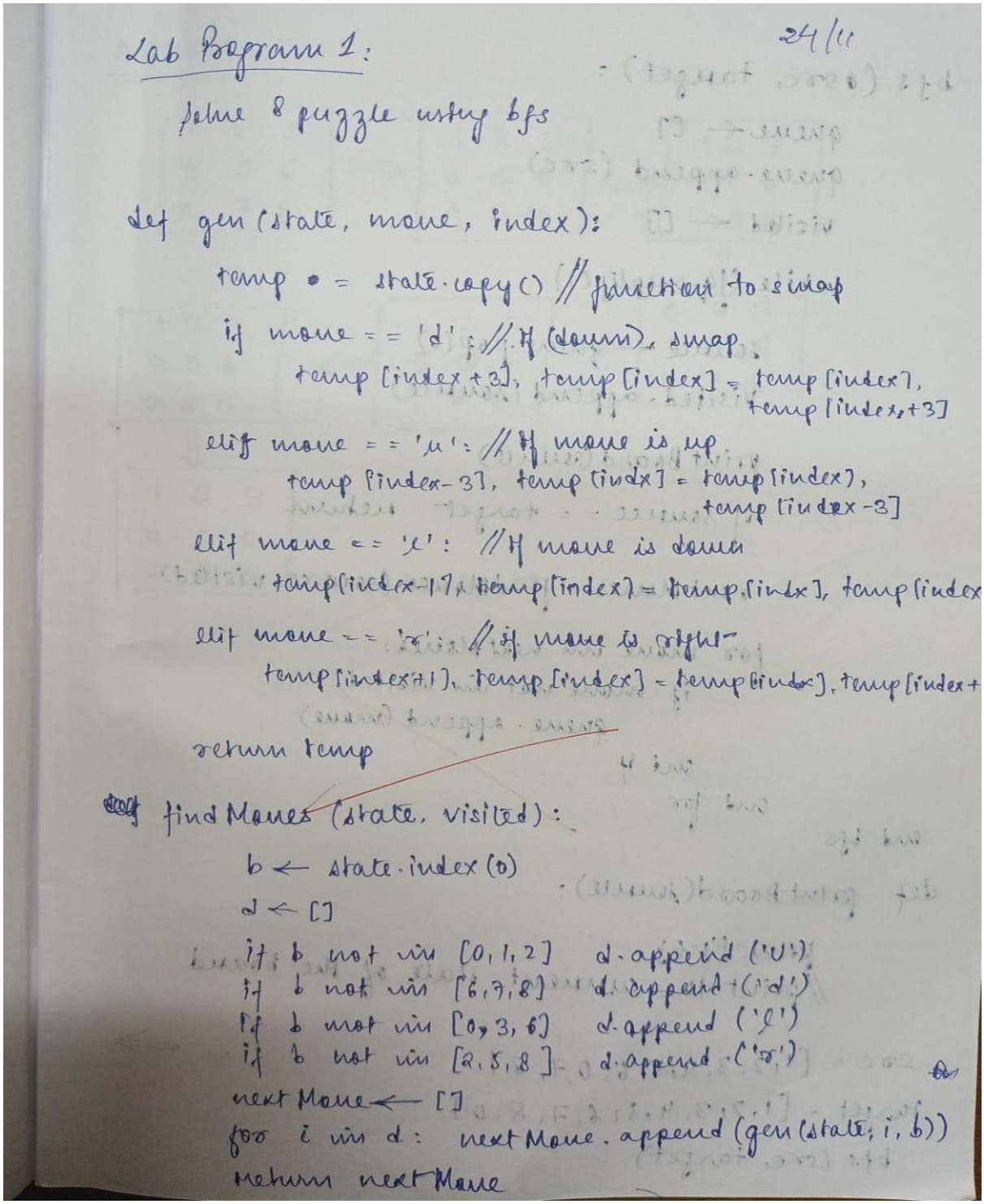

O O X
X X O
O X 9
computer's turn :
O O X
X X O
O X X

EXPERIMENT: - 2

Aim

Solve 8 puzzle problems.

Observation Notebook



Lab Program 1: 24/11

Solve 8 puzzle using bfs

```
def gen(state, move, index):
    temp = state.copy() // function to swap
    if move == 'd': // If (down), swap
        temp[index+3], temp[index] = temp[index],
                                     temp[index,+3]
    elif move == 'u': // If move is up
        temp[index-3], temp[indx] = temp[index),
                                    temp[index-3]
    elif move == 'l': // If move is down
        temp[index-1], temp[index] = temp.[indx], temp[index
    elif move == 'r': // if move is right
        temp[index+1], temp[index] = temp[indx], temp[index+
    return temp

def findMoves(state, visited):
    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')
    nextMove ← []
    for i in d: nextMove.append(gen(state, i, b))
    return nextMove
```

```
bfs (src, target):
    queue ← []
    queue.append (src)
    visited ← []

    while (!q.empty())
        source ← queue.pop(0)
        visited.append (source)

        printBoard (source)
        if source == target return

        nextMoves = findMoves (source, visited)

        for move in nextMoves:
            if move not in visited,
                queue.append (move)
            end if
        end for
end bfs

def printBoard(source):
    // print the current state of the board

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

Code

```
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

```
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
```

EXPERIMENT: - 3

Aim

Implement Iterative deepening search algorithm.

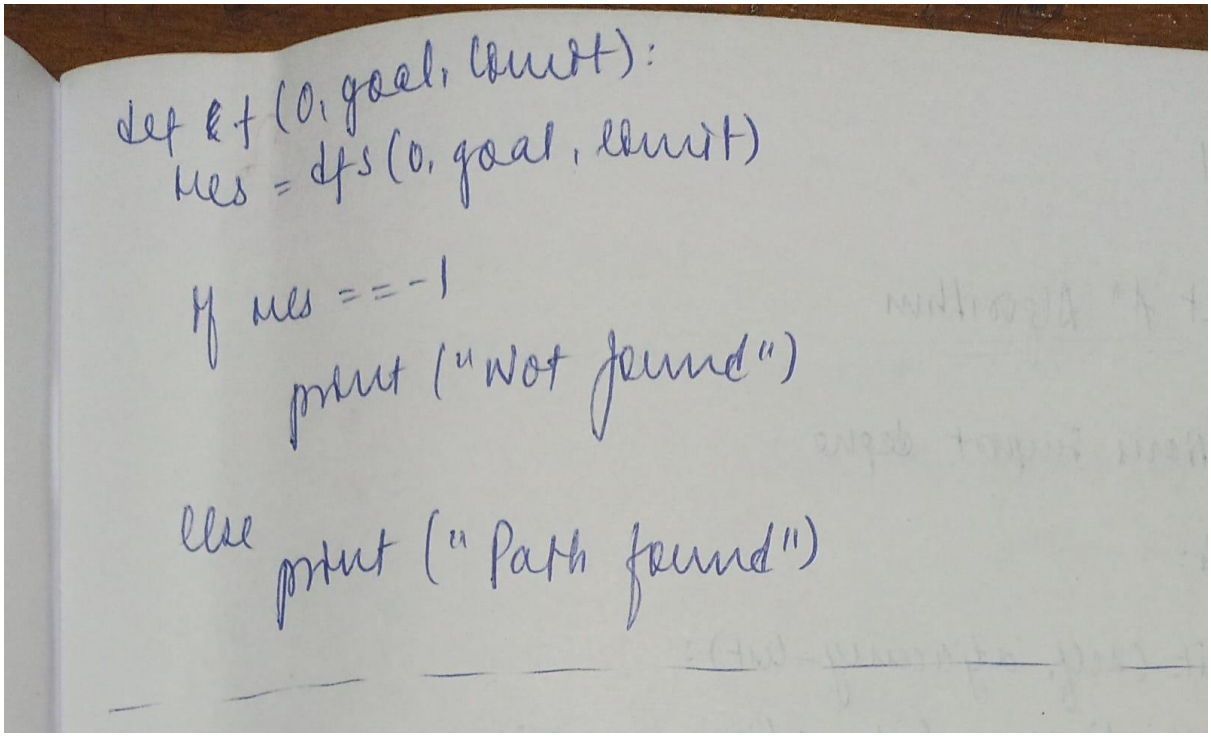
Observation Notebook

Program 3

Iterative Deepening Search

code:

```
from collections import defaultdict

n = input("no. of nodes")
e = input("no. of edges"):

graph = defaultdict(list)

for i in range(e):
    i,j = map(int, input().split()):
    graph[i].append(j)

def dfs(v, goal, limit):
    if (v = goal)
        return 1
    for i in graph[v]
        if limit >= 1:
            if dfs(i, goal, limit-1) != -1
                return 1

    return -1

goal = int("enter goal")
limit = int(input("enter limit"))
```



Code

```
  # 8 Puzzle problem using Iterative deepening depth first 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")
```

Output

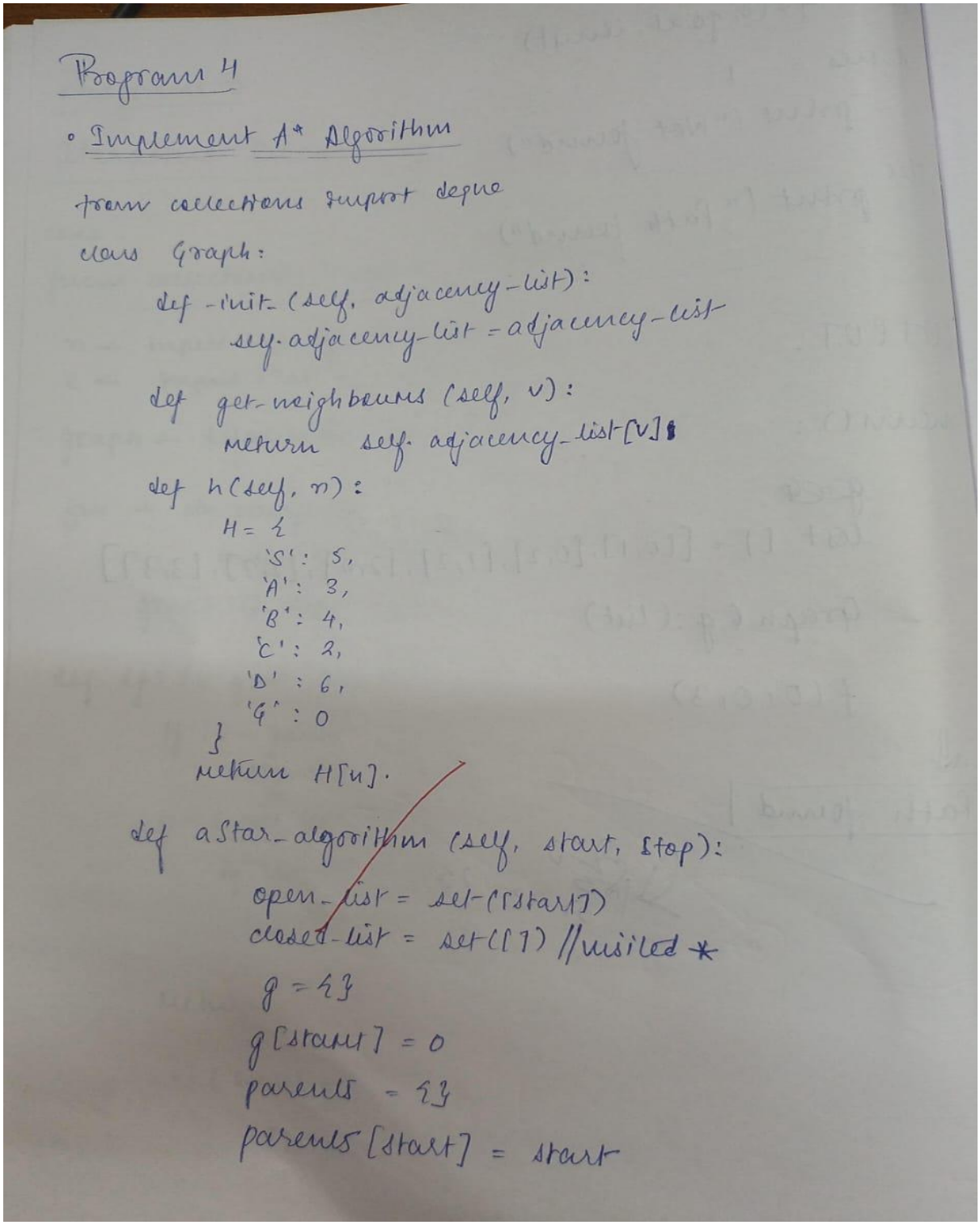
```
Success!! It is possible to solve 8 Puzzle problem
Path: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8], [1, 2, 3, 4, 5, 6, 7, 8, 0]]
```

EXPERIMENT: - 4

Aim

Implement A* search algorithm.

Observation Notebook

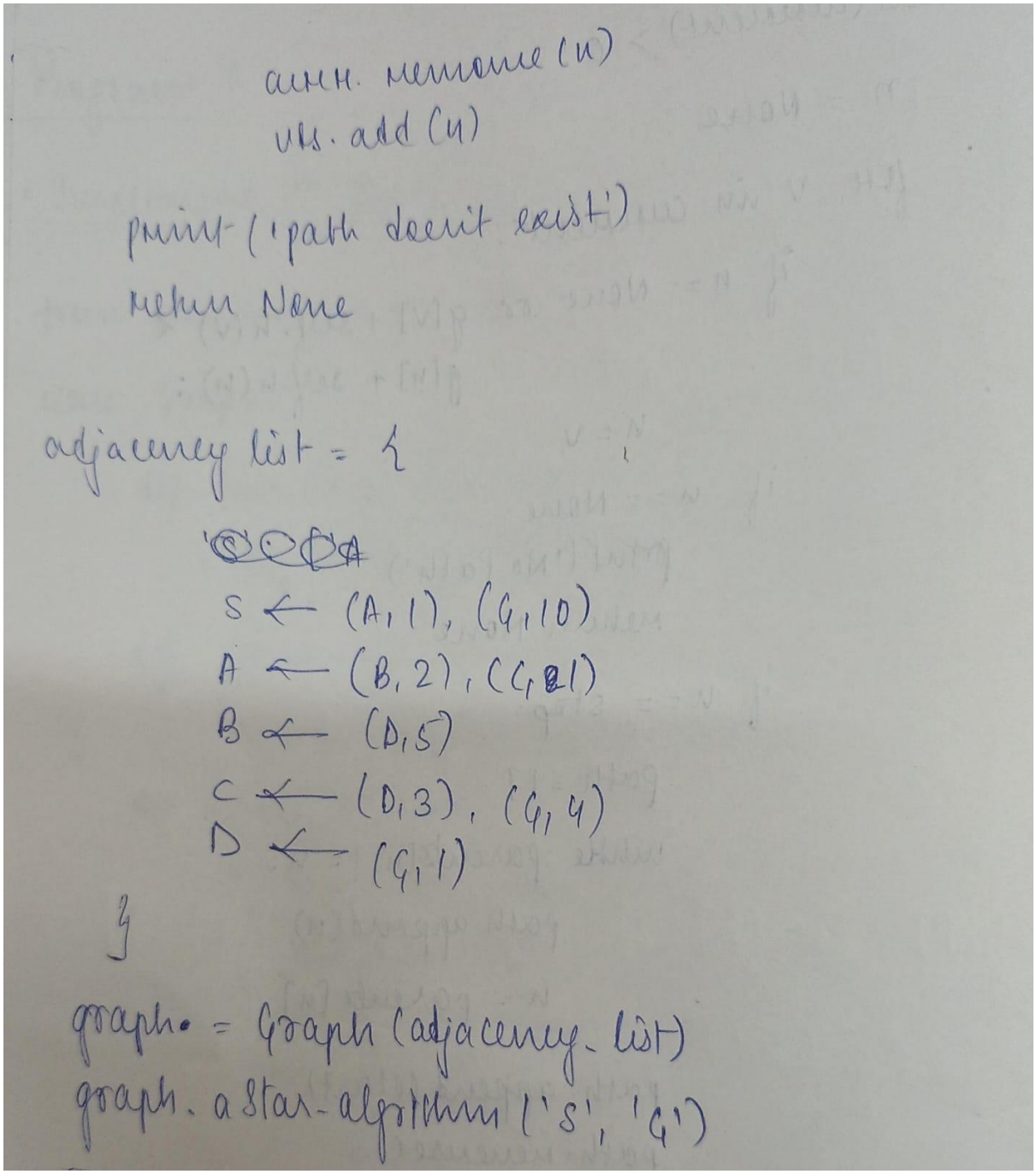


Program 4

- Implement A* Algorithm

```
from collections import deque
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbours(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {
            'S': 5,
            'A': 3,
            'B': 4,
            'C': 2,
            'D': 6,
            'G': 0
        }
        return H[n].
    def aStar_algorithm(self, start, stop):
        open_list = set([start])
        closed_list = set([]) //visited *
        g = {}
        g[start] = 0
        parents = {}
        parents[start] = start
```

```
while len(current) > 0:
    n = None
    for v in current:
        if n == None or g[v] + self.h(v) <
                        g[n] + self.h(n):
            n = v
    if n == None:
        print('No Path')
        return None
    if n == stop:
        path = []
        while parents[n] != n:
            path.append(n)
            n = parents[n]
        path.append(start)
        path.reverse()
        print('path found : %s' .format(path
        return path
    for (m, weight) in self.get_neighbours(n)
        if m not in cur and m not in v
            cur.add(m)
            parents[m] = n
            g[m] = g[n] + weight
        else:
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n
                if m in vis:
                    vis.remove(
                    cur.add(m)
```



Code

```
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.process()
```

Output

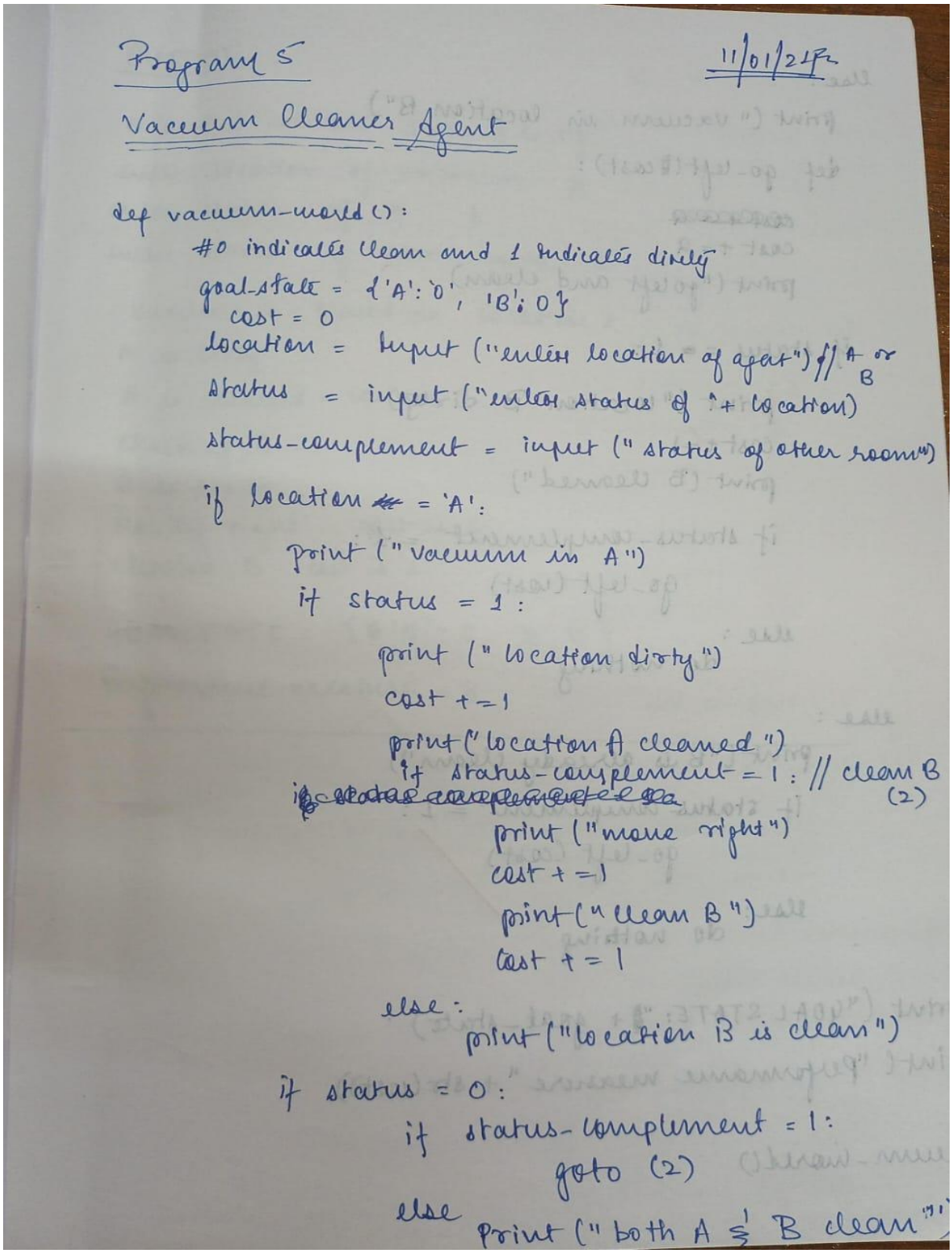
```
Enter the start state matrix

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 _
```

EXPERIMENT: - 5

Aim

Implement vacuum cleaner agent.

Observation Notebook



Program 5 11/01/24

Vacuum Cleaner Agent

```
def vacuum-world():
    #0 indicates clean and 1 indicates dirty
    goal_state = {'A':'0', 'B':'0'}
    cost = 0
    location = input("enter location of agent") // A or B
    status = input("enter status of " + location)
    status-complement = input("status of other room")
    if location == 'A':
        print("vacuum in A")
        if status = 1:
            print("location dirty")
            cost += 1
            print("location A cleaned")
            if status-complement = 1: // clean B
                print("move right")
                cost += 1
                print("clean B")
                cost += 1
        else:
            print("location B is clean")
    if status = 0:
        if status-complement = 1:
            goto (2)
        else
            print("both A & B clean")
```

```
else:
    print("vacuum in location B")
    def go_left(cost):
        cost += 2
        print("go left and clean")
    if status == 1:
        print("location B dirty")
        cost += 1
        print("B cleaned")
        if status_complement = 1:
            go_left(cost)
        else:
            do nothing
    else:
        print("B is already clean")
        if status_complement = 1:
            go_left(cost)
        else:
            do nothing
print("GOAL STATE:" + goal_state)
print("Performance measure" + str(cost))

vacuum_world()
```

Code

```
def vacuum_world():
    # Initializing goal_state for four rooms
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': 0, 'B': 0, 'C': 0, 'D': 0}
    cost = 0

    # User input for initial vacuum location and status of each room
    location_input = input("Enter Initial Location of Vacuum (A/B/C/D):
")
    print("Enter status of each room (1 - dirty, 0 - clean):")
    for room in goal_state:
        goal_state[room] = int(input(f"Status of Room {room}: "))

    print("Initial Location Condition: " + str(goal_state))

    # Function to clean a room
    def clean_room(room):
        nonlocal cost
        if goal_state[room] == 1:
            print(f"Cleaning Room {room}...")
            goal_state[room] = 0
            cost += 1  # Cost for cleaning
            print(f"Room {room} has been cleaned. Current cost:
{cost}")
        else:
            print(f"Room {room} is already clean.")

    # Cleaning logic
    rooms = ['A', 'B', 'C', 'D']
    current_index = rooms.index(location_input)

    # Clean all rooms starting from the initial location
    for i in range(current_index, len(rooms)):
        clean_room(rooms[i])

    # Clean remaining rooms (if the initial location was not 'A')
    for i in range(0, current_index):
        clean_room(rooms[i])

    # Output final state and performance measure
    print("Final State of Rooms: " + str(goal_state))
    print("Performance Measurement (Total Cost): " + str(cost+4))

vacuum_world()
```

Output

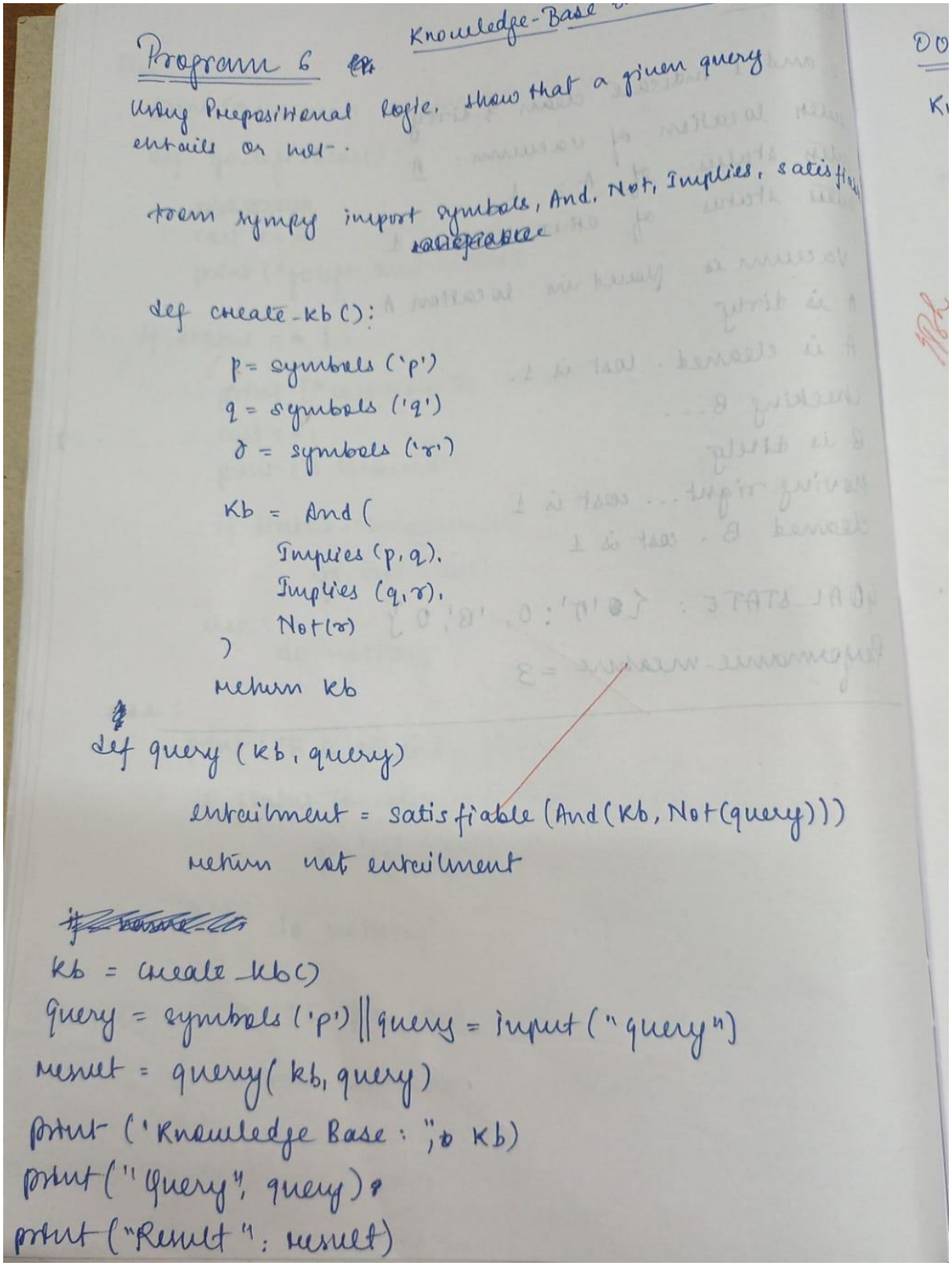
```
Enter clean status for Room 1 (1 for dirty, 0 for clean): 1
Enter clean status for Room 2 (1 for dirty, 0 for clean): 0
Cleaning Room 1 (Room was dirty)
Room 1 is now clean.
Room 2 is already clean.
Returning to Room 1 to check if it has become dirty again:
Room 1 is already clean.
Room 1 is clean after checking.

Enter clean status for Room at (1, 1) (1 for dirty, 0 for clean): 1
Enter clean status for Room at (1, 2) (1 for dirty, 0 for clean): 0
Enter clean status for Room at (2, 1) (1 for dirty, 0 for clean): 1
Enter clean status for Room at (2, 2) (1 for dirty, 0 for clean): 1
Cleaning Room at (1, 1) (Room was dirty)
Room is now clean.
Room at (1, 2) is already clean.
Cleaning Room at (2, 1) (Room was dirty)
Room is now clean.
Cleaning Room at (2, 2) (Room was dirty)
Room is now clean.
Returning to Room at (1, 1) to check if it has become dirty again:
Room at (1, 1) is already clean.
```

EXPERIMENT: - 6

Aim

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

Observation Notebook



Code

```
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

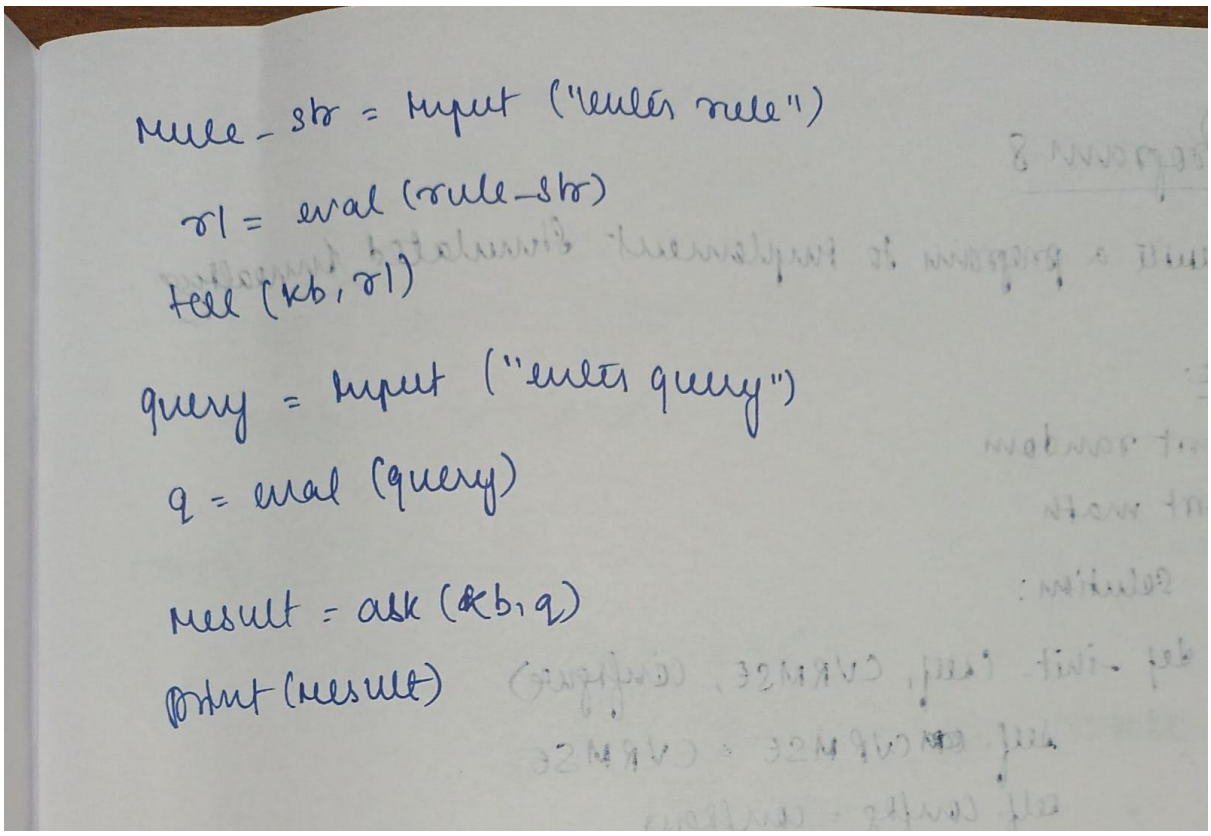
```
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

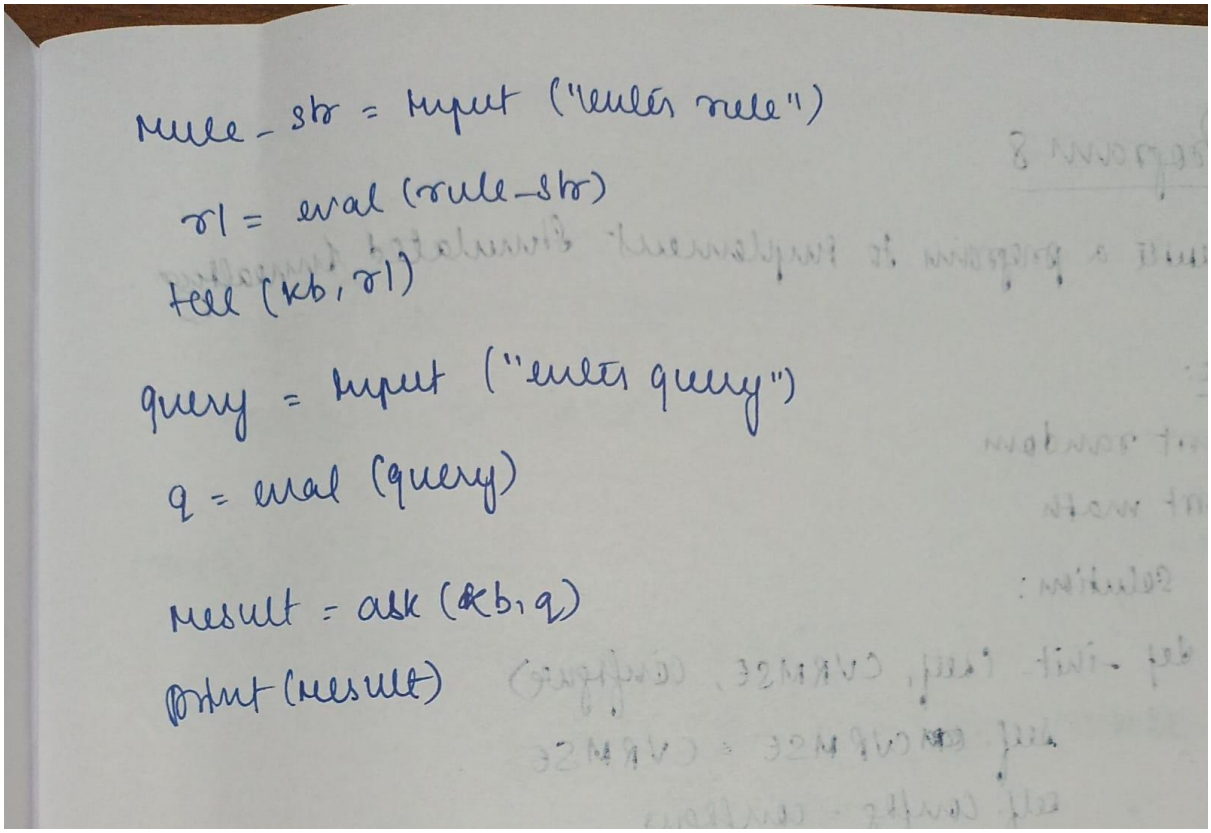
EXPERIMENT: - 7

Aim

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

Observation Notebook





Code

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
def negate(term):
    return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):
    if len(clause) > 2:
        t = split_terms(clause)
        return f'{t[1]}v{t[0]}'
    return ''
def split_terms(rule):
    exp = '(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms

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

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(goal)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if
contradiction(goal,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
                                steps[''] = f"Resolved {temp[i]} and
{temp[j]} to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when
{negate(goal)} is assumed as true. Hence, {goal} is true."
                                return steps
                    elif len(gen) == 1:
                        clauses += [f'{gen[0]}']
                    else:
                        if
contradiction(goal,f'{terms1[0]}v{terms2[0]}'):
                            temp.append(f'{terms1[0]}v{terms2[0]}')
                            steps[''] = f"Resolved {temp[i]} and
{temp[j]} to {temp[-1]}, which is in turn null. \
                            \nA contradiction is found when
{negate(goal)} is assumed as true. Hence, {goal} is true."
                            return steps
            for clause in clauses:
                if clause not in temp and clause != reverse(clause) and
reverse(clause) not in temp:
                    temp.append(clause)
                    steps[clause] = f'Resolved from {temp[i]} and
{temp[j]}.'
            j = (j + 1) % n
        i += 1
    return steps
rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
goal = 'R'
main(rules, goal)
```

Output

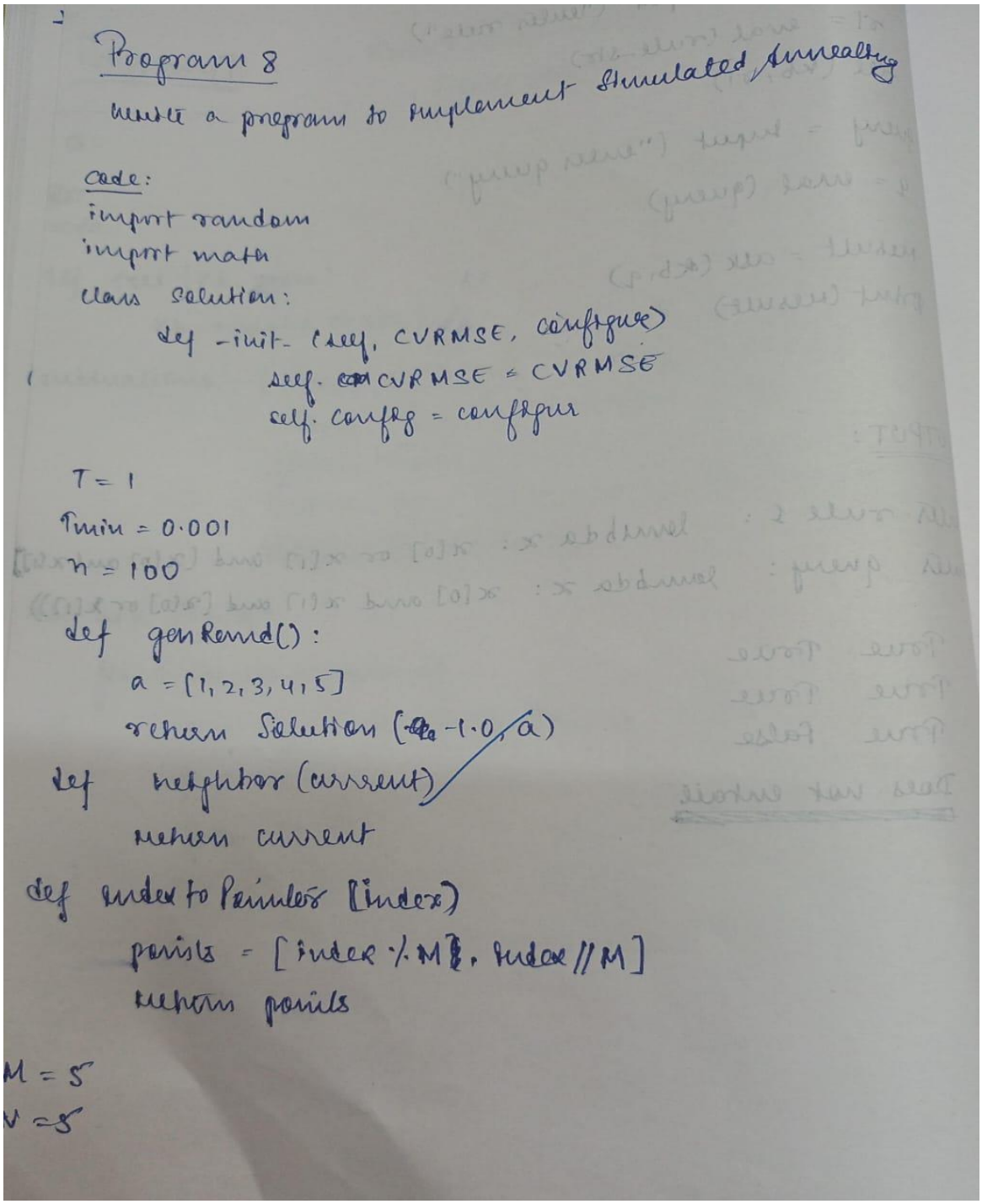
```
Step    |Clause |Derivation
-------------------------------
 1.     | Rv~P  | Given.
 2.     | Rv~Q  | Given.
 3.     | ~RvP  | Given.
 4.     | ~RvQ  | Given.
 5.     | ~R    | Negated conclusion.
 6.     |       | Resolved Rv~P and ~RvP to Rv~R, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

EXPERIMENT: - 8

Aim

Write a program to implement Simulated Annealing Algorithm

Observation Notebook



Program 8

Write a program to implement Simulated Annealing

Code:

```
import random
import math
class Solution:
    def __init__(self, CVRMSE, configure)
        self.CVRMSE = CVRMSE
        self.config = configur

T = 1
Tmin = 0.001
n = 100
def genRand():
    a = [1,2,3,4,5]
    return Solution(-1.0, a)
def neighbor(current)
    return current
def indexToPoints(index)
    points = [index % M, index // M]
    return points

M = 5
N = 5
```

```
Source = [ ['X' for range in (N)]
           for j in range[M]]

min = Solution (float('inf'), None)
Current = getRand()

while T > Tmin:
    for i in range (num):
        if curr.CVRMSE < min.CVRMSE:
            min = curr
```

// Display output
print (SourceArray)

OUTPUT

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

Code

```
import random
import math

class Solution:
    def __init__(self, CVRMSE, configuration):
        self.CVRMSE = CVRMSE
        self.config = configuration

# Function prototype
def gen_rand_sol():
    # Instantiating for the sake of compilation
    a = [1, 2, 3, 4, 5]
    return Solution(-1.0, a)

# Global variables
T = 1
Tmin = 0.0001
alpha = 0.9
num_iterations = 100
M = 5
N = 5
source_array = [['X' for _ in range(N)] for _ in range(M)]
temp = []
mini = Solution(float('inf'), temp)
current_sol = gen_rand_sol()

def neighbor(current_sol):
    return current_sol

def cost(input_configuration):
    return -1.0

# Mapping from [0, M*N] --> [0,M]x[0,N]
```

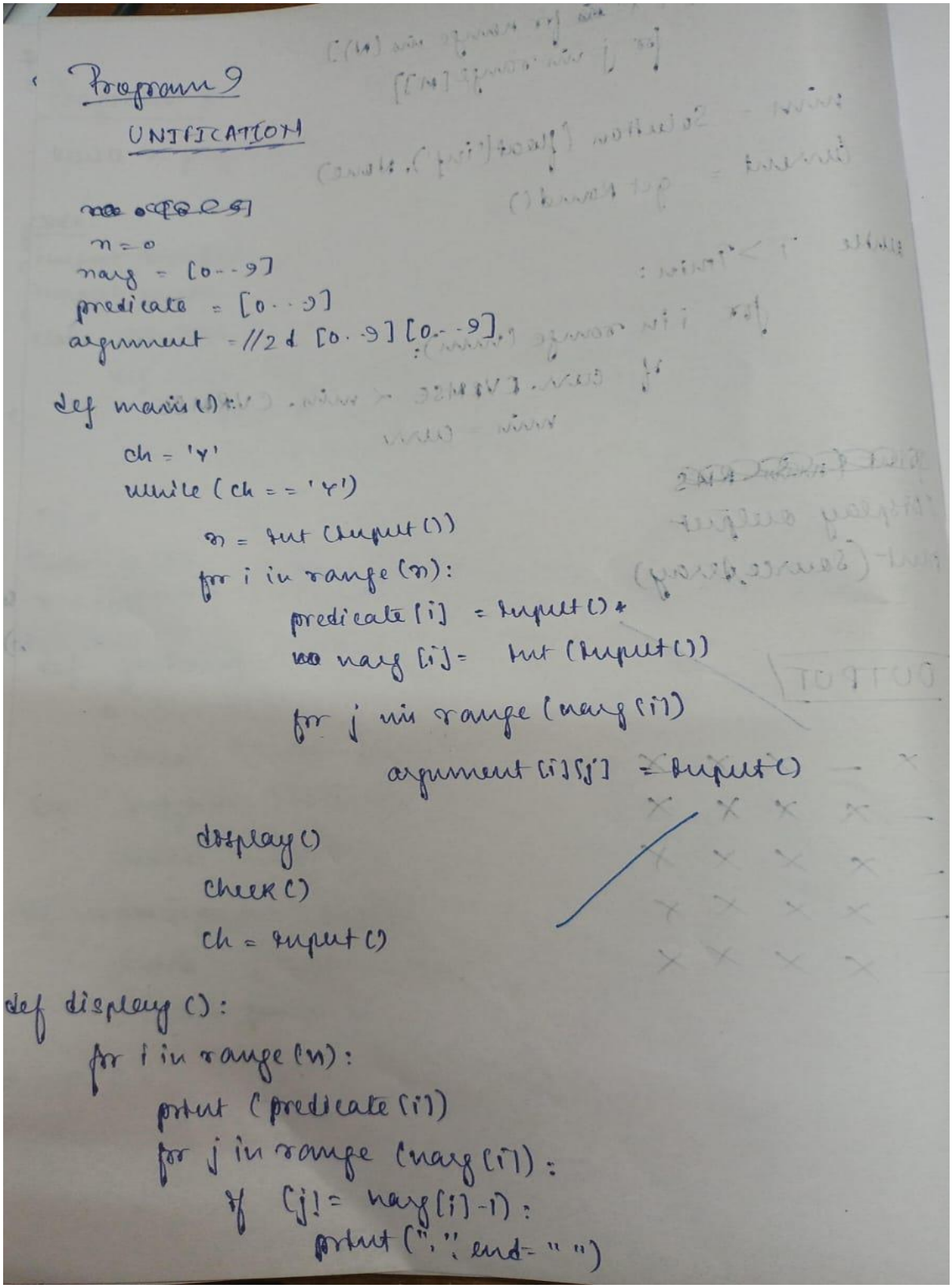
```
def index_to_points(index):
    points = [index % M, index // M]
    return points

while T > Tmin:
    for _ in range(num_iterations):
        # Reassigns global minimum accordingly
        if current_sol.CVRMSE < mini.CVRMSE:
            mini = current_sol
        new_sol = neighbor(current_sol)
        ap = math.exp((current_sol.CVRMSE - new_sol.CVRMSE) / T)
        if ap > random.random():
            current_sol = new_sol
    T *= alpha  # Decreases T, cooling phase

print(mini.CVRMSE, "\n\n")

for i in range(M):
    for j in range(N):
        source_array[i][j] = 'X'

# Displays
for index in range(len(mini.config)):
    obj = mini.config[index]
    coord = index_to_points(obj)
    source_array[coord[0]][coord[1]] = '-'

# Displays optimal location
for i in range(M):
    row = ""
    for j in range(N):
        row = row + source_array[i][j] + " "
    print(row)
```

EXPERIMENT: - 9

Aim

Implement unification in first order logic

Observation Notebook



Program 9

UNIFICATION

```
n = 0
narg = [0--9]
predicate = [0..9]
argument = //2 d [0..9] [0--9]

def main():
    ch = 'Y'
    while (ch == 'Y')
        n = int (input())
        for i in range (n):
            predicate [i] = input()
            narg [i] = int (input())
            for j in range (narg [i])
                argument [i][j] = input()
        display()
        check()
        ch = input()

def display():
    for i in range (n):
        print (predicate [i])
        for j in range (narg [i]):
            if (j != narg [i] - 1):
                print (",", end = " ")
```

```
def unify():
    flag = 0
    for i in range (n-1):
        for j in range (len(i))
            if (argument[i][j] != argument[i+1][j]
                if (!flag):
                    print argument[i+1][j]/"arg[i][j]
                    flag+1
    if (!flag)
        print ("error")

def check():
    pflag=0, aflag=0
    for i in range (n-1):
        if (predicate[i] != predicate[i+1])
            pflag=1
            break
    if (pflag!=1)
        ind=0
        key = narg[ind]
        l = len (narg)
        for i in range [0, key-1):
            if (i>= key): continue
            if (ind! = l-1):
                ind+=1
```

Code

```
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression
def getInitialPredicate(expression):
    return expression.split("(")[0]

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

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

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

def apply(exp, substitutions):
    for substitution in substitutions:
        new, old = substitution
        exp = replaceAttributes(exp, old, new)
    return exp
def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True


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


def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

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

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

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

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

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

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

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

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

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

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

    remainingSubstitution = unify(tail1, tail2)
    if not remainingSubstitution:
        return False

    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution
exp1 = "knows(X)"
exp2 = "knows(Richard)"
substitutions = unify(exp1, exp2)
print("Substitutions:")
print(substitutions)
```

Output

(for given input)

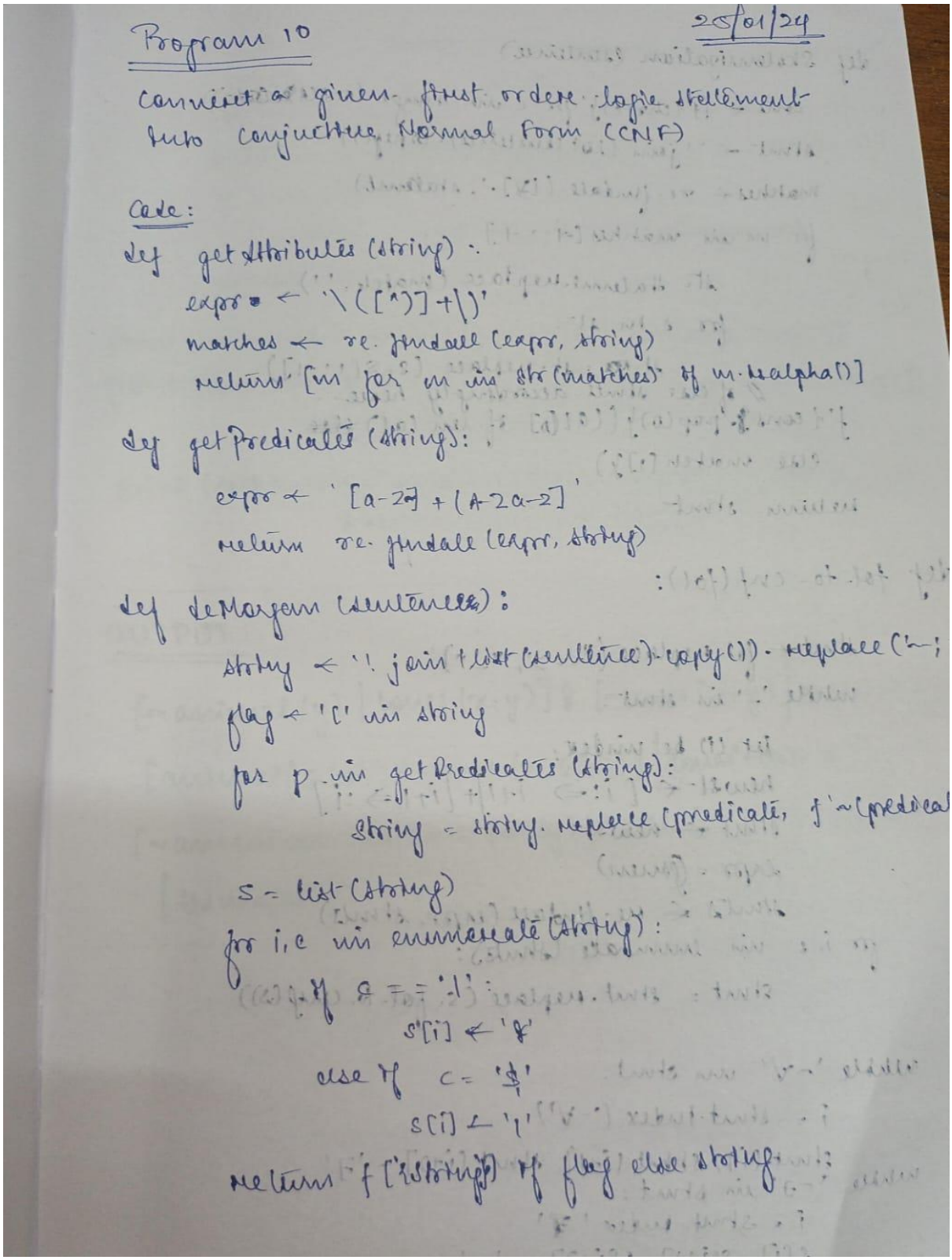
```
Substitutions:
[('X', 'Richard')]
```

EXPERIMENT: - 10

Aim

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

Observation Notebook



Program 10 — 25/01/24

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

Code:

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

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

def deMorgan(sentence):
    string = ''.join(list(sentence).copy()).replace('~~','')
    flag = '[' in string
    for p 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] = '|'
    return ''.join(s)[1:-1] if flag else ''.join(s)
```

```
def Skolemization(sentence)
    cons ← [f'{chr(c)}' for c in range(ord('A'), ord('A')+2)]
    stmt ← ''.join(list(sentence)).copy()
    matches ← re.findall('[∀].', statement)
    for m in matches[::-1]:
        st = statement.replace(match, '')
        for s in st:
            st = st.replace(s, s[1:-1])
        // if else statements accordingly hence--
    f'{cons.pop(0)}({a[0] if len(a) else
      else match[1]})
    return stmt

def fol_to_cnf(fol):
    stmt ← fol.replace("<=>", "_")
    while '_' in stmt:
        let (i) be index:
        new_stmt ← [i:⇒ i+1] + [i+1⇒ :i]
        stmt ← new_stmt
        expr = (fol, sep)
        stmts ← re.findall(expr, stmts)
    for i, s in enumerate(stmts):
        stmt = stmt.replace(s, fol_NOT_cnf(s))

while '~∀' in stmt:
    i = stmt.index('~∀')
    stmt[i], stmt[i+1], stmt[i+2] = '∃'
while '~∃' in stmt:
    i = stmt.index('∃')
    s[i], s[i+1], s[i+2] = '~'
```

```
strut ← strut.replace ('~[∀', '∃[~∀'),
                      ('~[∃', '[~∃')
expr ← '(~[∀|∃)'
strut = re.findall (expr, strut)
for s in strut:
        strut = strut.replace (s, DeMorgan(s))
return strut.
```

```
print (conv (fol-to-cnf ("animal(y) ⟺ loves (x,y)"))
print (conv (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:

[~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)

Code

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

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

def 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[love
s(z,x)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>crim
inal(x)"))
```

Output

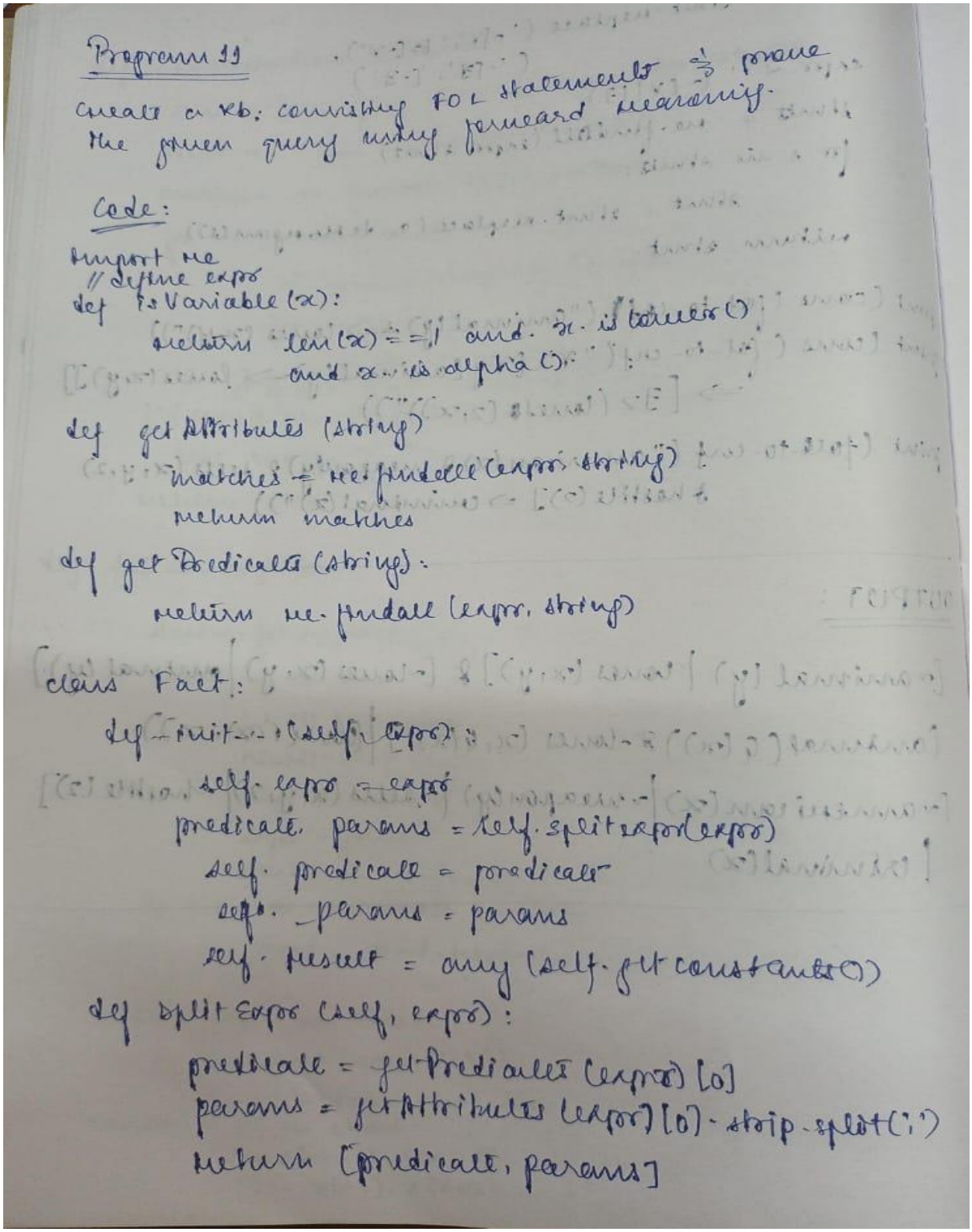
```
[~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
[animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

EXPERIMENT: - 11

Aim

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

Observation Notebook



```
def get_constants(self):
    return [name or c];
def getVariables(self):
    return [v for v in variables()]
def substitute(self, constants)
    c = copy of constants
    return fact()

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

class KB:
    // defining a knowledge base
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell():
        if '=>' in e:
            // implication
        else // add it to the facts

    def query(self, e):
        facts = set(), i=1
        for f in facts():
            if fact in predicate:
                i+=1
```

```
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 ('american(x) & weapon(y) & sells(x,y,z)
        & hostile(z) => criminal(x)')
kb.query ('criminal(x)')
kb.display()
```

OUTPUT

Querying Criminals:

1. criminal(west).

All facts:

1. criminal(west)
2. hostile(Nono)
3. weapon(M1)
4. missile(M1)
5. sells(west, M1, Nono)
6. enemy(Nono, America)
7. owns(Nono, M1)
8. American(west)

Code

```
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

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