VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

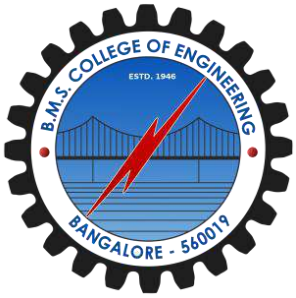
Artificial Intelligence

Submitted by:

Sanchay Agrawal (1BM21CS186)

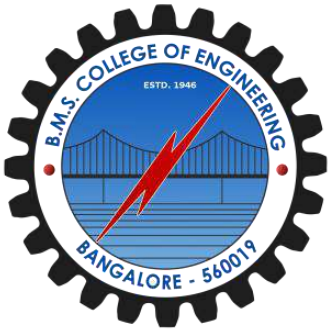
Under the Guidance of
Dr. Madhavi R.P.
Associate Professor, BMSCE

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



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



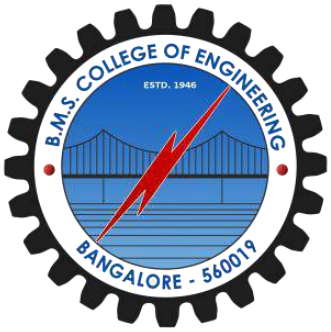
CERTIFICATE

This is to certify that the Lab work entitled "**Artificial Intelligence**" carried out by **Sanchay Agrawal (1BM21CS186),** 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 - (22CS5PCAIN)** work prescribed for the said degree.

Dr. Madhavi R.P.
Associate Professor
Department of CSE
BMSCE, Bengaluru

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

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



DECLARATION

I, Sanchay Agrawal (1BM21CS186), 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. Madhavi R.P., Associate 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

1. Course Outcomes

CO1: Apply knowledge of agent architecture, searching and reasoning techniques for different applications.

CO2: Analyse Searching and Inferencing Techniques.

CO3: Design a reasoning system for a given requirement

CO4: Conduct practical experiments for demonstrating agents, searching and inferencing.

2. Experiments

2.1 Experiment - 1

2.1.1 Question:

Implement Vacuum cleaner agent.

2.1.2 Code:

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

    location_input = input("Enter Location of Vacuum")
    status_input = int(input("Enter status of " + location_input))
    status_input_complement = int(input("Enter status of the other room"))
    print("Initial Location Condition" + str(goal_state))

    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
            goal_state['A'] = 0
            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 Location B. ")
                cost += 1  # Cost for moving right
                print("COST for moving RIGHT " + str(cost))
                # Suck the dirt and mark it as clean
```

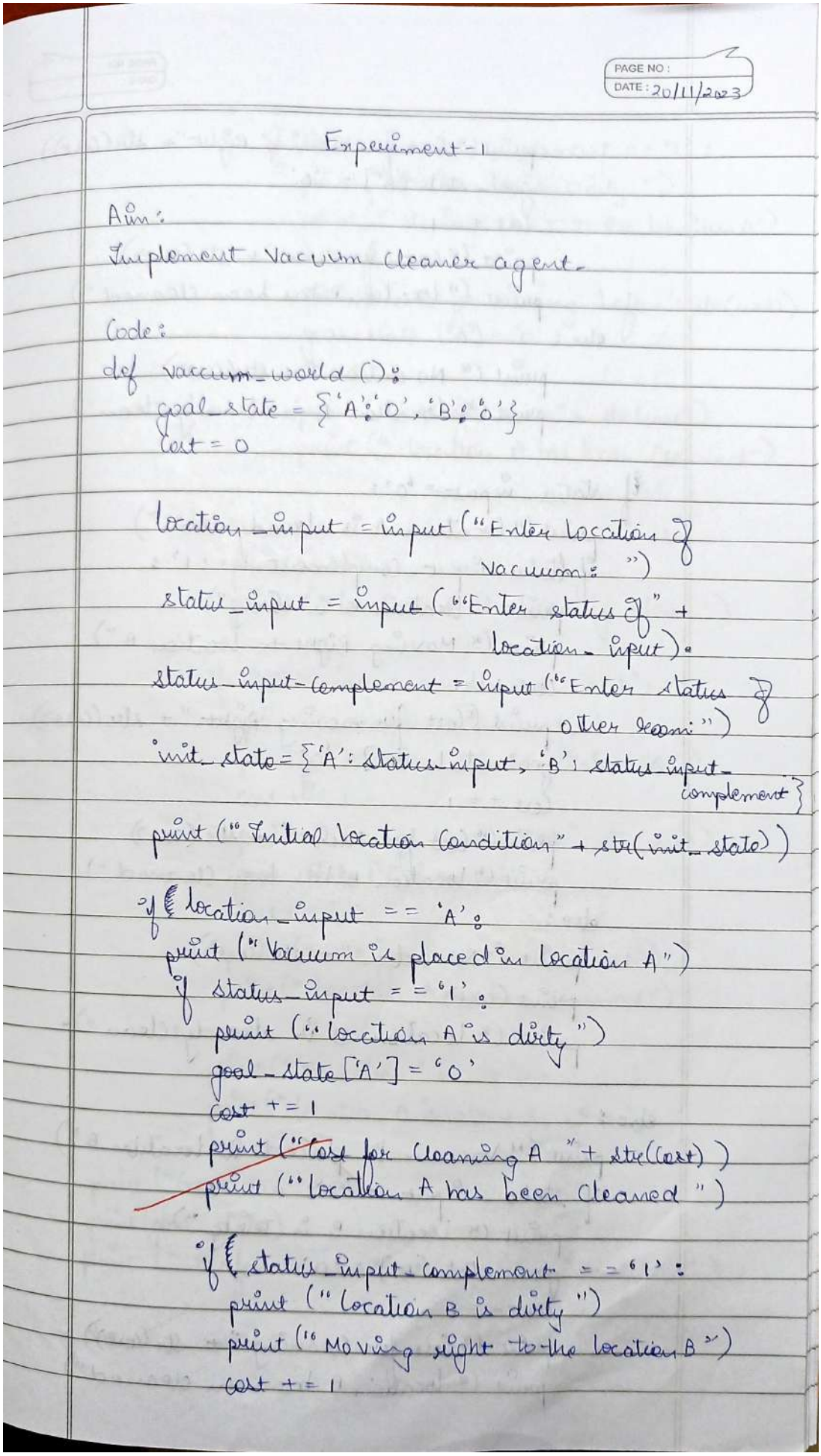
```
            goal_state['B'] = 0
            cost += 1  # Cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # Suck and mark clean
            print("Location B is already clean.")

    if status_input == 0:
        print("Location A is already clean ")
        if status_input_complement == 1:  # If B is Dirty
            print("Location B is Dirty.")
            print("Moving RIGHT to Location B. ")
            cost += 1  # Cost for moving right
            print("COST for moving RIGHT " + str(cost))
            # Suck the dirt and mark it as clean
            goal_state['B'] = 0
            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
        goal_state['B'] = 0
        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 Location A. ")
            cost += 1  # Cost for moving right
            print("COST for moving LEFT " + str(cost))
            # Suck the dirt and mark it as clean
            goal_state['A'] = 0
```

```
            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 Location A. ")
                cost += 1  # Cost for moving right
                print("COST for moving LEFT " + str(cost))
                # Suck the dirt and mark it as clean
                goal_state['A'] = 0
                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))


vacuum_world()
```

2.1.3 Output:

```
Enter Location of Vacuum: A
Enter status of A1
Enter status of the other room: 1
Initial Location Condition{'A': 0, 'B': 0}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to Location B.
COST for moving RIGHT 2
COST for SUCK 3
Location B has been Cleaned.
GOAL STATE:
{'A': 0, 'B': 0}
Performance Measurement: 3
```

2.1.4 Observation Book Pictures:



PAGE NO :
DATE : 20/11/2023

Experiment - 1

Aim:
Implement Vacuum cleaner agent.

Code:

```
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("Vacuum 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 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("Vacuum is placed in location B")
    if status_input == '1':
        print("location B is Dirty")
        goal_state['B'] = '0'
        cost += 1
        print("Cost for cleaning" + str(cost))
        print("location B has been cleaned")
```

```
    if status_input_complement == '1':
        print("location A is Dirty")
        print("Moving LEFT to the location A")
        cost += 1
        print("Cost for Moving left" + str(cost))
        goal_state['A'] = '0'
        cost += 1
        print("Cost for Suck" + str(cost))
        print("location A has been cleaned")

else:
    print(cost)
    print("location B is already clean")

    if status_input_complement == '1':
        print("location A is Dirty")
        print("Moving Left to 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()
```

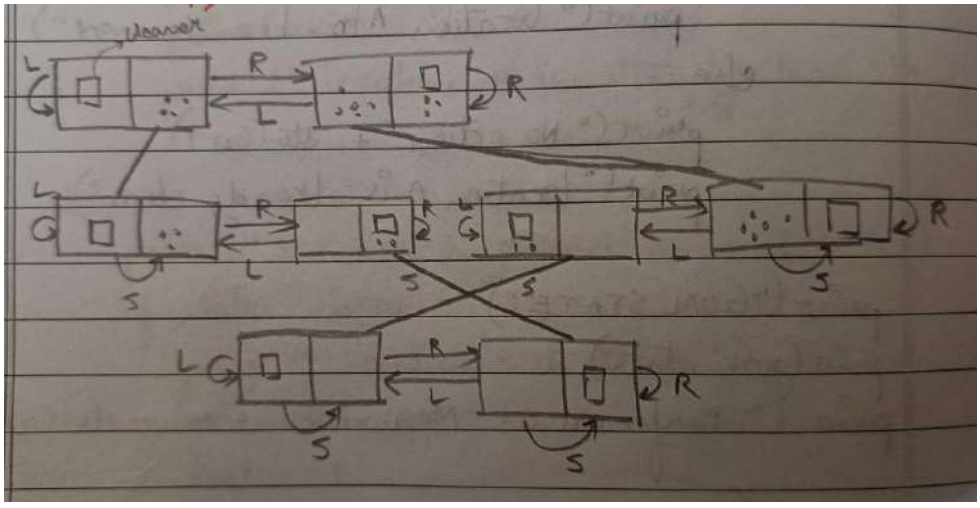
Output:

Enter location of Vacuum : A
Enter status of A : 1
Enter status of other room : 1
Initial location Condition : {'A': '1', 'B' : '1'}
Vacuum is placed in location A
location A is Dirty
Cost of cleaning A : 1
location A has been cleaned
location B is Dirty.
Moving Right to location B.
Cost for Moving Right : 2
Cost for Suck : 3
location B has been cleaned.

Goal State:
{'A': '0', 'B' : '0'}

Performance Measurement : 3

20/11/23



2.2 Experiment - 2

2.2.1 Question:

Implement 8 puzzle using Breadth First Search Algorithm.

2.2.2 Code:

```
import sys
import numpy as np

class Node:
def __init__(self, state, parent, action):
self.state = state
self.parent = parent
self.action = action


class StackFrontier:
def __init__(self):
self.frontier = []

def add(self, node):
self.frontier.append(node)

def contains_state(self, state):
return any((node.state[0] == state[0]).all() for node in self.frontier)

def empty(self):
return len(self.frontier) == 0

def remove(self):
if self.empty():
raise Exception("Empty Frontier")
else:
node = self.frontier[-1]
self.frontier = self.frontier[:-1]
return node


class QueueFrontier(StackFrontier):
def remove(self):
if self.empty():
raise Exception("Empty Frontier")
else:
node = self.frontier[0]
self.frontier = self.frontier[1:]
return node
```

```
class Puzzle:
def __init__(self, start, startIndex, goal, goalIndex):
self.start = [start, startIndex]
self.goal = [goal, goalIndex]
self.solution = None

def neighbors(self, state):
mat, (row, col) = state
results = []

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

return results

def print(self):
solution = self.solution if self.solution is not None else None
print("Start State:\n", self.start[0], "\n")
print("Goal State:\n",  self.goal[0], "\n")
print("\nStates Explored: ", self.num_explored, "\n")
print("Solution:\n ")
for action, cell in zip(solution[0], solution[1]):
print("action: ", action, "\n", cell[0], "\n")
print("Goal Reached!!")

def does_not_contain_state(self, state):
for st in self.explored:
```

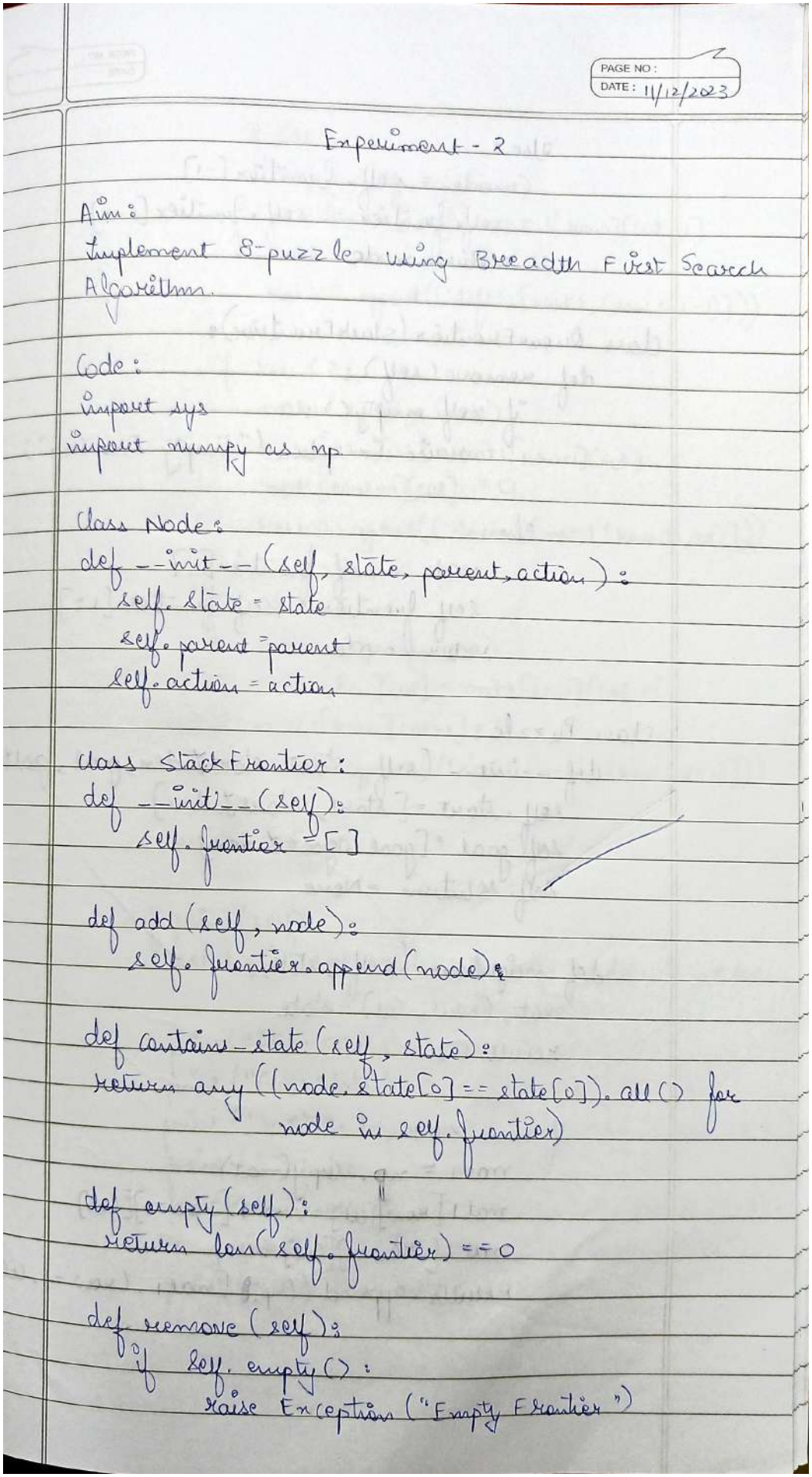
```
if (st[0] == state[0]).all():
return False
return True

def solve(self):
self.num_explored = 0

start = Node(state=self.start, parent=None, action=None)
frontier = QueueFrontier()
frontier.add(start)

self.explored = []

while True:
if frontier.empty():
raise Exception("No solution")

node = frontier.remove()
self.num_explored += 1

if (node.state[0] == self.goal[0]).all():
actions = []
cells = []
while node.parent is not None:
actions.append(node.action)
cells.append(node.state)
node = node.parent
actions.reverse()
cells.reverse()
self.solution = (actions,  cells)
return

self.explored.append(node.state)

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


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


startIndex = (1, 1)
goalIndex = (1, 0)
```

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

2.2.3 Output:

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

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


States Explored:  358

Solution:

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

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

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

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

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

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

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

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

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

Goal Reached!!
```

2.2.4 Observation Book Pictures:



PAGE NO :
DATE : 11/12/2023

Experiment - 2

Aim:
Implement 8-puzzle using Breadth First Search Algorithm.

Code:

```
import sys
import numpy as np

class Node:
def __init__(self, state, parent, action):
    self.state = state
    self.parent = parent
    self.action = action

class StackFrontier:
def __init__(self):
    self.frontier = []

def add(self, node):
    self.frontier.append(node)

def contains_state(self, state):
return any((node.state[0] == state[0]).all() for
                node in self.frontier)

def empty(self):
return len(self.frontier) == 0

def remove(self):
    if self.empty():
        raise Exception("Empty Frontier")
```

```
        else:
            node = self.frontier[-1]
            self.frontier = self.frontier[:-1]
            return node.

class QueueFrontier(StackFrontier):
    def remove(self):
        if self.empty():
            raise Exception("Empty Frontier")

        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node.

class Puzzle:
    def __init__(self, start, startIndex, goal, goalIndex):
        self.start = [start, startIndex]
        self.goal = [goal, goalIndex]
        self.solution = None.

    def neighbours(self, state):
        mat, (row, col) = state
        results []

        if row > 0:
            mat1 = np.copy(mat)
            mat1[row][col] = mat1[row-1][col]
            mat1[row-1][col] = 0
            results.append(('up', [mat1, (row-1, col)]))
```

DATE :

```
if col > 0:
    mat1 = np.copy(mat)
    mat1[row][col] = mat1[row][col-1]
    mat1[row][col-1] = 0
    results.append(('left', [mat1, (row, col-1)]))

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

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

return results

def print(self):
    solutions = self.solution if self.solution is not None
                              else None
    print("Start State:\n", self.start[0], "\n")
    print("Goal State:\n", self.goal[0], "\n")
    print("\n States Explored:", self.num_explored, "\n")
    print("Solutions:\n")
    for action, cell in zip(solution[0], solution[1]):
        print("action:", action, "\n", cell[0], "\n")
    print("Goal Reached")
```

```
def does_not_contain_state(self, state):
    for st in self.explored:
        if (st[0] == state[0]).all():
            return False
    return True

def solve(self):
    self.num_explored = 0

    start = Node(state=self.start, parent=None,
                 action=None)
    frontier = QueueFrontier()
    frontier.add(start)

    self.explored = []

    while True:
        if frontier.empty():
            raise Exception("No solution")

        node = frontier.remove()
        self.num_explored += 1

        if (node.state[0] == self.goal[0]).all():
            actions = []
            cells = []
            while node.parent is not None:
                actions.reverse()
                cells.reverse()
                self
                actions.append(node.action)
                cells.append(node.state)
                node = node.parent
```

```
        actions.reverse()
        cells.reverse()
        self.solution = (actions, cells)
        return

    self.explored.append(node.state)

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

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

startIndex = (1,1)
goalIndex = (1,0)

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

Output:

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

PAGE NO :
DATE :

Goal State :

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

Solution :

action : up

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

action : left

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

action : down

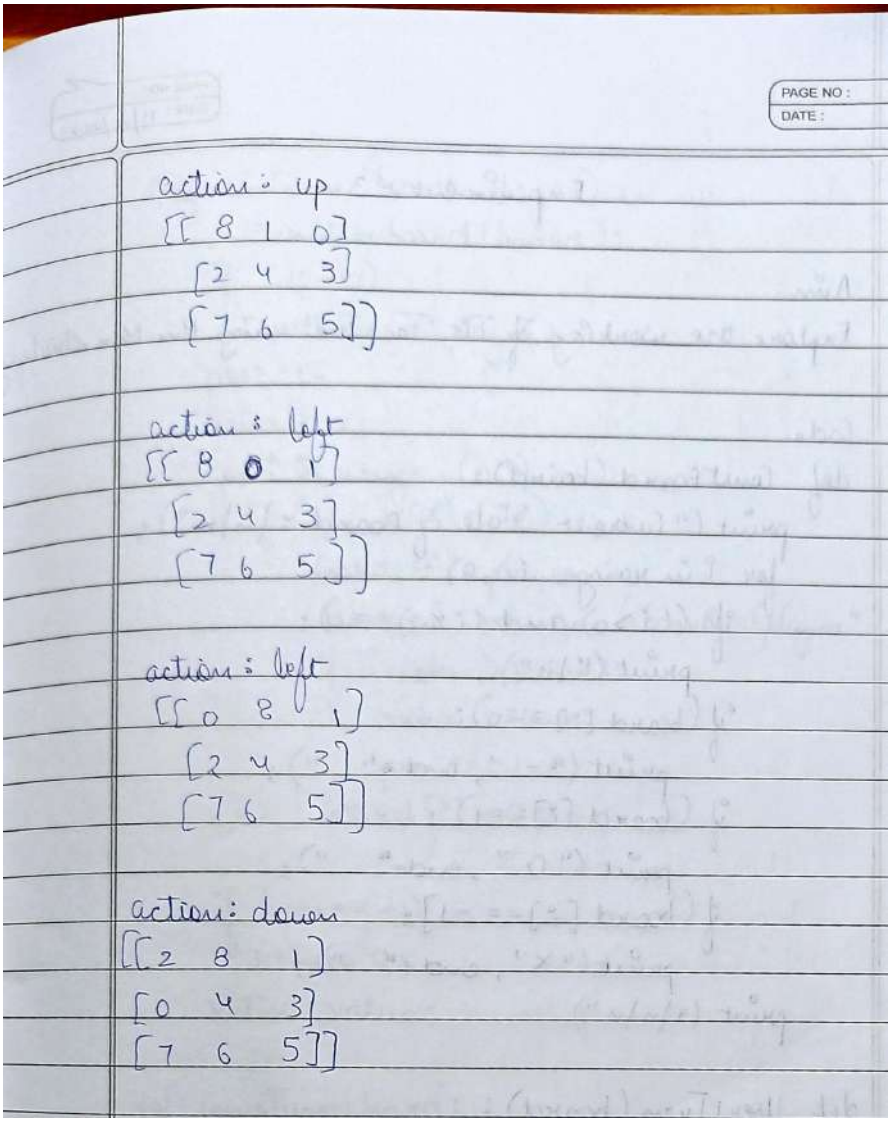
```
[[8 1 3]
 [0 2 4]
 [7 6 5]]
```

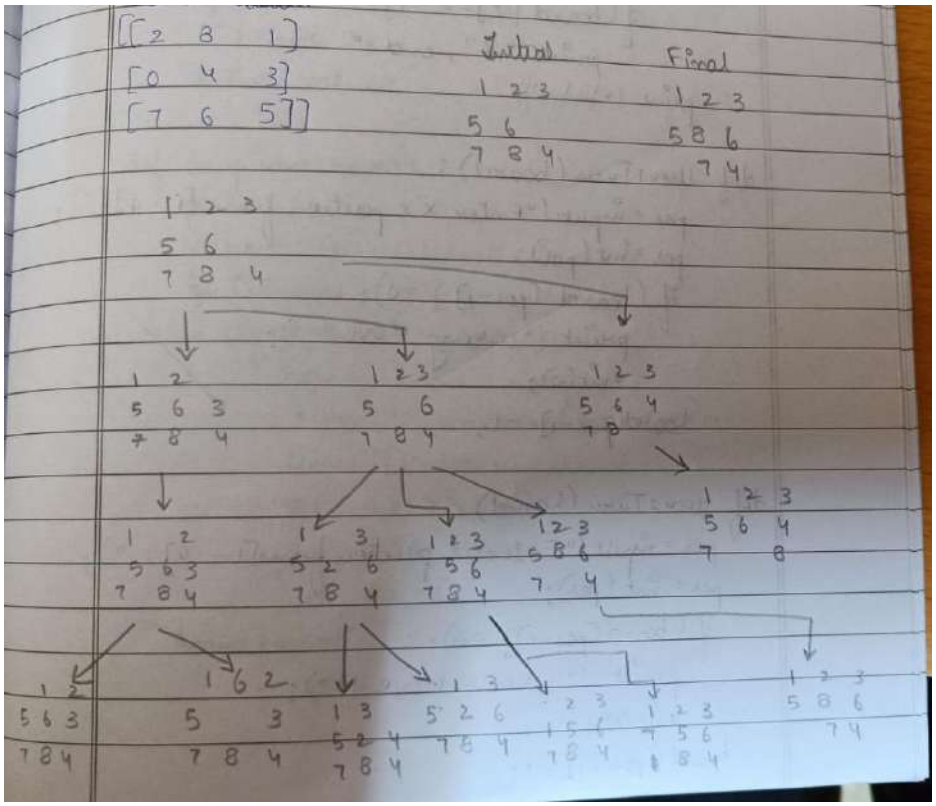
action : right

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

action : right

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


PAGE NO :
DATE :
action : up
[[8 1 0]
[2 4 3]
[7 6 5]]
action : left
[[8 0 1]
[2 4 3]
[7 6 5]]
action : left
[[0 8 1]
[2 4 3]
[7 6 5]]
action : down
[[2 8 1]
[0 4 3]
[7 6 5]]


Initial
Final

2.3 Experiment - 3

2.3.1 Question:

Explore the working of Tic-tac-toe using Min Max strategy.

2.3.2 Code:

```
def ConstBoard(board):
    print("Current State Of Board : \n\n");
    for i in range (0,9):
        if((i>0) and (i%3)==0):
            print("\n");
        if(board[i]==0):
            print("- ",end=" ");
        if (board[i]==1):
            print("O ",end=" ");
        if(board[i]==-1):
            print("X ",end=" ");
    print("\n\n");

#This function takes the user move as input and make the required changes on the board.
def User1Turn(board):
    pos=input("Enter X's position from [1...9]: ");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!!");
        exit(0) ;
    board[pos-1]=-1;

def User2Turn(board):
    pos=input("Enter O's position from [1...9]: ");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!!");
        exit(0);
    board[pos-1]=1;

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

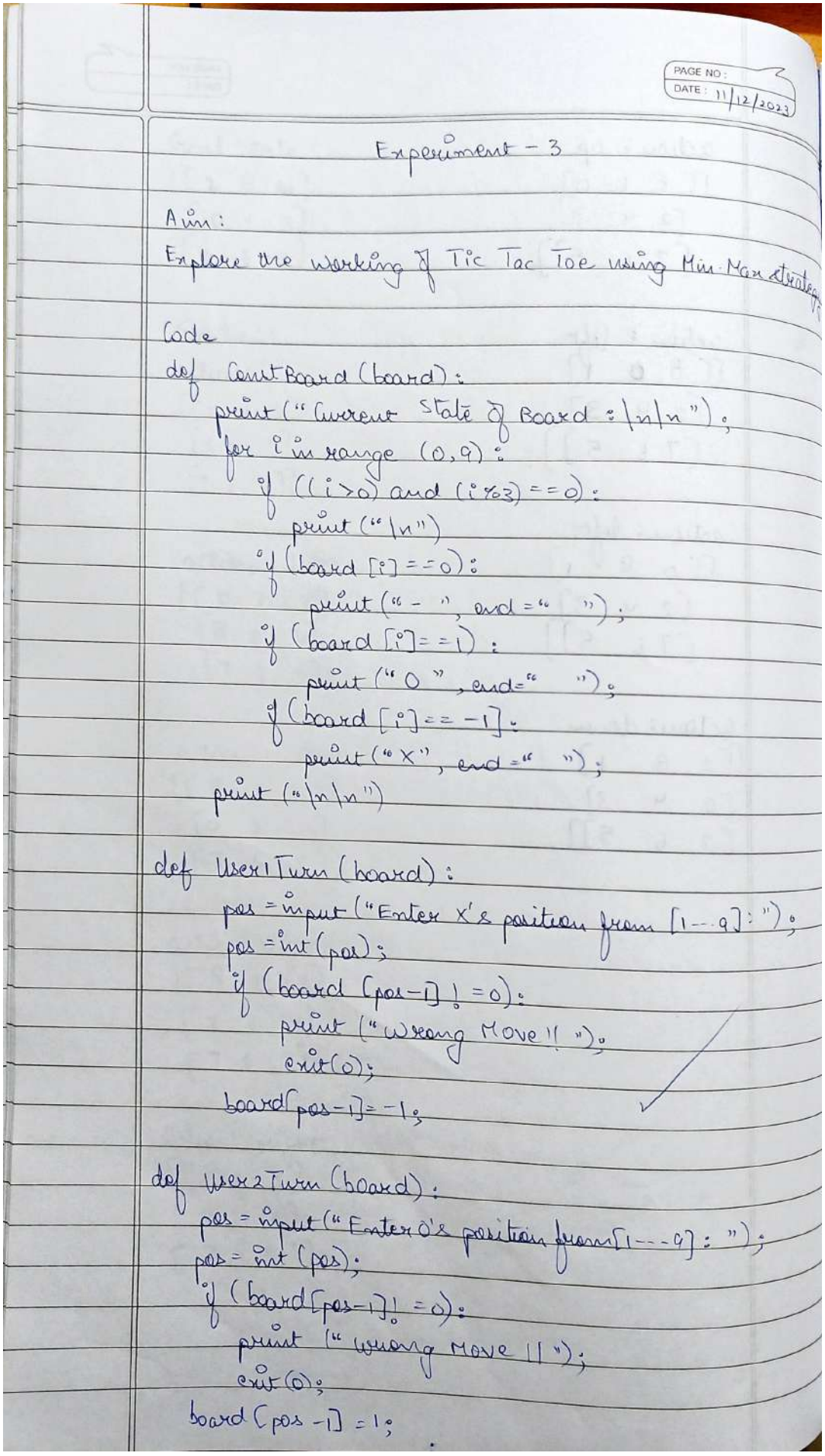
```
            if(score>value):
                value=score;
                pos=i;
            board[i]=0;

    if(pos==-1):
        return 0;
    return value;

#This function makes the computer's move using minmax algorithm.
def CompTurn(board):
    pos=-1;
    value=-2;
    for i in range(0,9):
        if(board[i]==0):
            board[i]=1;
            score=-minimax(board, -1);
            board[i]=0;
            if(score>value):
                value=score;
                pos=i;

    board[pos]=1;


#This function is used to analyze a game.
def analyzeboard(board):
    cb=[[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]];

    for i in range(0,8):
        if(board[cb[i][0]] != 0 and
           board[cb[i][0]] == board[cb[i][1]] and
           board[cb[i][0]] == board[cb[i][2]]):
            return board[cb[i][2]];
    return 0;

#Main Function.
def main():
    choice=input("Enter 1 for single player, 2 for multiplayer: ");
    choice=int(choice);
    #The broad is considered in the form of a single dimentional array.
    #One player moves 1 and other move -1.
    board=[0,0,0,0,0,0,0,0,0];
    if(choice==1):
        print("Computer : O Vs. You : X");
        player= input("Enter to play 1(st) or 2(nd) :");
```

```
        player = int(player);
        for i in range (0,9):
            if(analyzeboard(board)!=0):
                break;
            if((i+player)%2==0):
                CompTurn(board);
            else:
                ConstBoard(board);
                User1Turn(board);
    else:
        for i in range (0,9):
            if(analyzeboard(board)!=0):
                break;
            if((i)%2==0):
                ConstBoard(board);
                User1Turn(board);
            else:
                ConstBoard(board);
                User2Turn(board);


    x=analyzeboard(board);
    if(x==0):
        ConstBoard(board);
        print("Draw!!!")
    if(x==-1):
        ConstBoard(board);
        print("X Wins!!! Y Loose !!!")
    if(x==1):
        ConstBoard(board);
        print("X Loose!!! O Wins !!!!")

main()
```

2.3.3 Output:

```
Enter 1 for single player, 2 for multiplayer: 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :1
Current State Of Board :


-  -  -

-  -  -

-  -  -


Enter X's position from [1...9]: 1
Current State Of Board :


X  -  -

-  O  -

-  -  -


Enter X's position from [1...9]: 7
Current State Of Board :


X  -  -

O  O  -

X  -  -


Enter X's position from [1...9]: 6
Current State Of Board :


X  O  -

O  O  X

X  -  -


Enter X's position from [1...9]: 3
Current State Of Board :


X  O  X

O  O  X

X  O  -


X Loose!!! O Wins !!!!
```

2.3.4 Observation Book Pictures:



PAGE NO :
DATE : 11/12/2023

Experiment - 3

Aim:
Explore the working of Tic Tac Toe using Min-Max strategy

Code

```
def CurrBoard(board):
    print("Current State of Board :\n\n");
    for i in range(0,9):
        if((i>0) and (i%3)==0):
            print("\n")
        if(board[i]==0):
            print("- ", end=" ");
        if(board[i]==1):
            print("O ", end=" ");
        if(board[i]==-1):
            print("X", end=" ");
    print("\n\n")

def User1Turn(board):
    pos=input("Enter X's position from [1--9]:");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!");
        exit(0);
    board[pos-1]=-1;

def User2Turn(board):
    pos=input("Enter O's position from[1--9]:");
    pos=int(pos);
    if(board[pos-1]!=0):
        print("Wrong Move!!");
        exit(0);
    board[pos-1]=1;
```

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

    if ( pos == -1)
        return 0;
    return value;

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

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

    for i in range(0,8):
        if (board[cb[i][0]] != 0 and
            board[cb[i][0]] == board[cb[i][1]] and
            board[cb[i][0]] == board[cb[i][2]]):
                return board[cb[i][2]];
    return 0;

def main():
    choice = input("Enter 1 for single player, 2 for
                    multiplayer : ");
    choice = int(choice);
    board = [0,0,0,0,0,0,0,0,0];
    if (choice == 1):
        print("Computer : O Vs You: X");
        player = input("Enter to play 1st or 2nd : ");
        player = int(player);
        for i in range(0,9):
            if (analyzeboard(board) != 0):
                break;
            if ((i+player)%2 == 0):
                CompTurn(board);
            else:
                InitBoard(board);
                User1Turn(board);

    else:
        for i in range(0,9):
            if (analyzeboard(board) != 0):
                break;
```

```
        if ((i)%2 == 0):
            ConstBoard(board);
            User1Turn(board);
        else:
            ConstBoard(board);
            User2 Turn(board);

    x = analyzeboard(board);
    if (x == 0):
        ConstBoard(board);
        print("Draw")
    if (x == -1):
        ConstBoard(board);
        print("X wins ! O losses !)
    if (x == 1):
        ConstBoard(board);
        print("X looses ! O Wins ! ").

main()
```

Output

```
Enter 1 for single player, 2 for multiplayer : 1
Computer : O Vs You : X
Enter to play 1st or 2nd : 1
Current state of Board:
- - -
- - -
- - -

Enter x's position from [1 - 9] : 1
```

Current State of Board:

X _ _

_ O _

_ _ _

Enter X's position from [1 - 9]: 7

Current State of Board:

X _ _

O O _

X _ _

Enter X's position from [1 - 9]: 6

Current State of Board:

X O _

O O X

X _ _

Enter X's position from [1 - 9]: 3

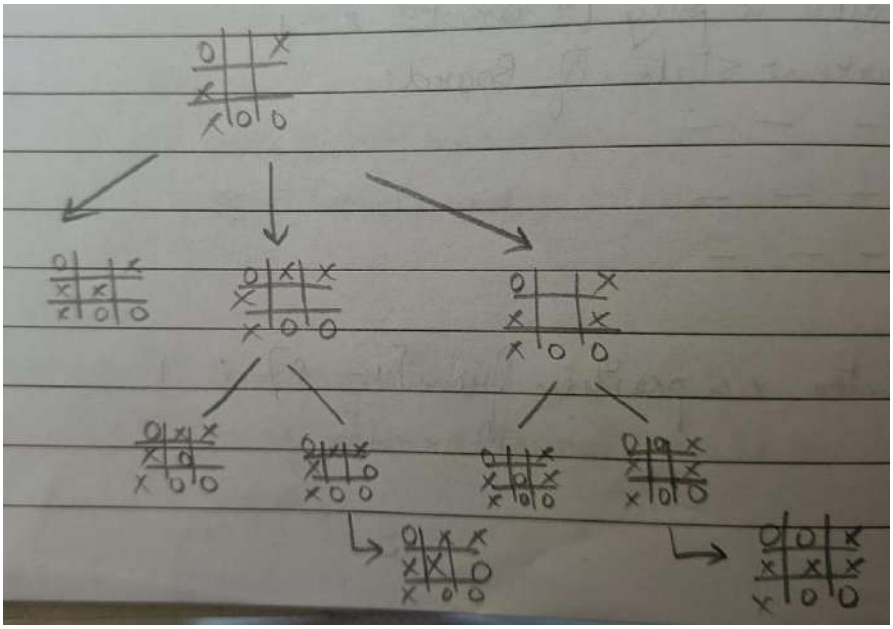
Current State of Board:

X O X

O O X

X O _

X loses!! O wins!!



2.4 Experiment - 4

2.4.1 Question:

Implement Iterative deepening search.

2.4.2 Code:

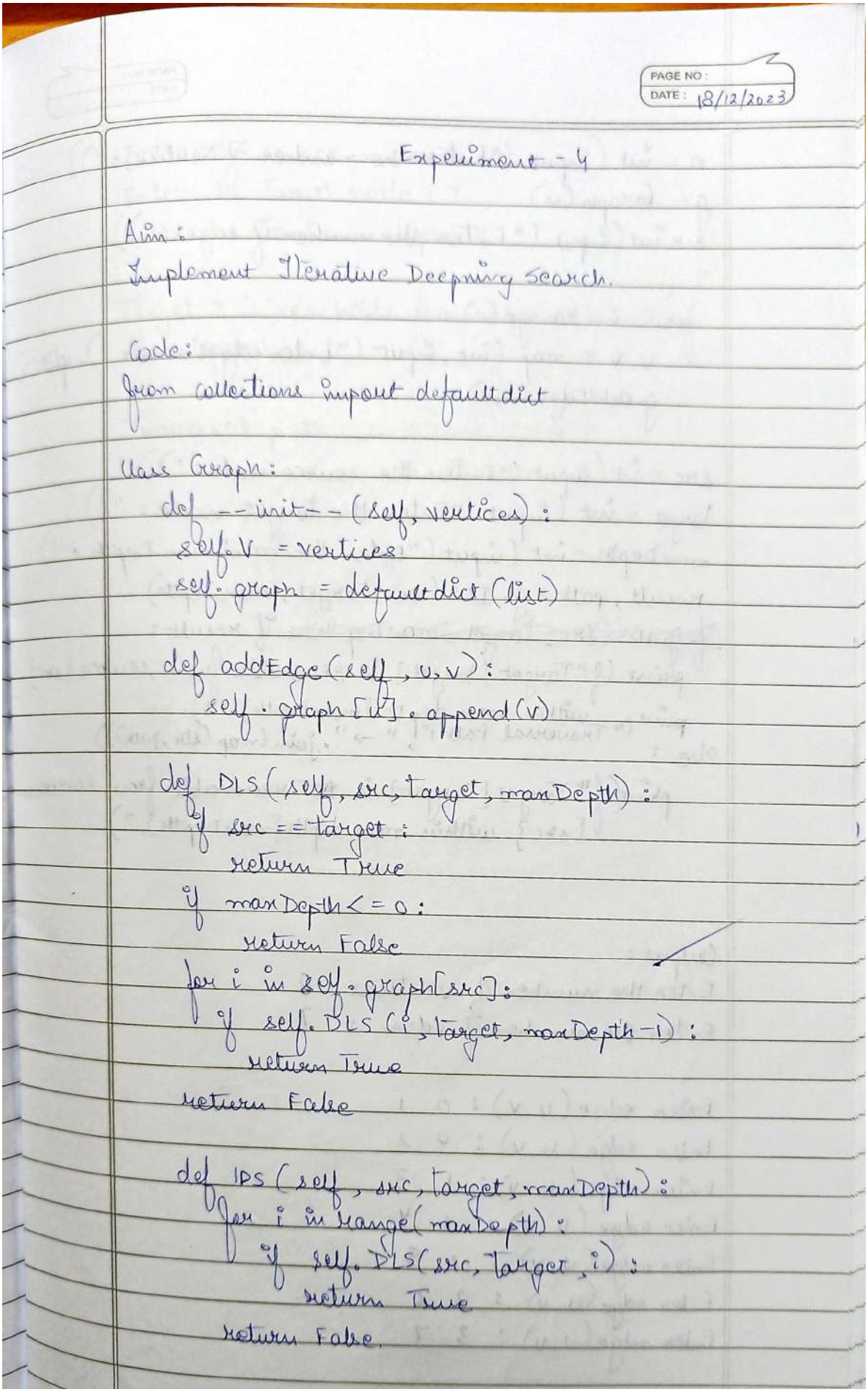
```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DLS(self, src, target, maxDepth, path):
        path.append(src)  # Append the current node to the path
        if src == target:
            return True, path
        if maxDepth <= 0:
            return False, path
        for i in self.graph[src]:
            result, path = self.DLS(i, target, maxDepth - 1, path)
            if result:
                return True, path
        path.pop()  # Remove the current node from the path if the target is not found at this
level
        return False, path

    def IDDFS(self, src, target, maxDepth):
        for i in range(maxDepth):
            path = []  # Initialize an empty path for each iteration
            result, path = self.DLS(src, target, i, path)
            if result:
                return True, path
        return False, []

# Create a graph based on user input
n = int(input("Enter the number of vertices: "))
g = Graph(n)

e = int(input("Enter the number of edges: "))
for _ in range(e):
    u, v = map(int, input("Enter edge (u v): ").split())
    g.addEdge(u, v)
```

```
src = int(input("Enter the source node: "))
target = int(input("Enter the target node: "))
maxDepth = int(input("Enter the maximum depth: "))

# Perform IDDFS and check if the target is reachable from the source within the max depth
result, path = g.IDDFS(src, target, maxDepth)

if result:
    print(f"Target {target} is reachable from source {src} within max depth {maxDepth}.")
    print("Traversal Path:", " -> ".join(map(str, path)))
else:
    print(f"Target {target} is NOT reachable from source {src} within max depth
{maxDepth}.")
```

2.4.3 Output:

```
Enter the number of vertices: 8
Enter the number of edges: 7
Enter edge (u v): 0 1
Enter edge (u v): 0 2
Enter edge (u v): 1 3
Enter edge (u v): 1 4
Enter edge (u v): 2 5
Enter edge (u v): 2 6
Enter edge (u v): 3 7
Enter the source node: 0
Enter the target node: 7
Enter the maximum depth: 4
Target 7 is reachable from source 0 within max depth 4.
Traversal Path: 0 -> 1 -> 3 -> 7
```

2.4.4 Observation Book Pictures:



PAGE NO :
DATE : 18/12/2023

Experiment - 4

Aim :
Implement Iterative Deepning search.

Code :

```
from collections import defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DLS(self, src, target, maxDepth):
        if src == target:
            return True
        if maxDepth <= 0:
            return False
        for i in self.graph[src]:
            if self.DLS(i, target, maxDepth-1):
                return True
        return False

    def IDS(self, src, target, maxDepth):
        for i in range(maxDepth):
            if self.DLS(src, target, i):
                return True
        return False
```

```
n = int(input("Enter the number of vertices: "))
g = Graph(n)
e = int(input("Enter the number of edges: "))

for _ in range(e):
    u, v = map(int, input("Enter edge(u v): ").split())
    g.addEdge(u,v)

src = int(input("Enter the source node: "))
Target = int(input("Enter the target node: "))
maxDepth = int(input("Enter the maximum Depth: "))
result, path = g.IDS(src, target, maxDepth)
if result:
    print(f"Target {target} is reachable from source {src}
    print("within max depth {maxDepth}.")
    "Traversal Path:", " → ".join(map(str,path)))
else:
    print(f"Target {Target} is NOT reachable from source
        {src} within max depth {maxDepth}.")
```

Output:

```
Enter the number of vertices : 8
Enter the number of edges : 7

Enter edge(u v): 0 1
Enter edge(u v): 0 2
Enter edge(u v): 1 3
Enter edge(u v): 1 4
Enter edge(u v): 2 5
Enter edge(u v): 2 6
Enter edge(u v): 3 7
```

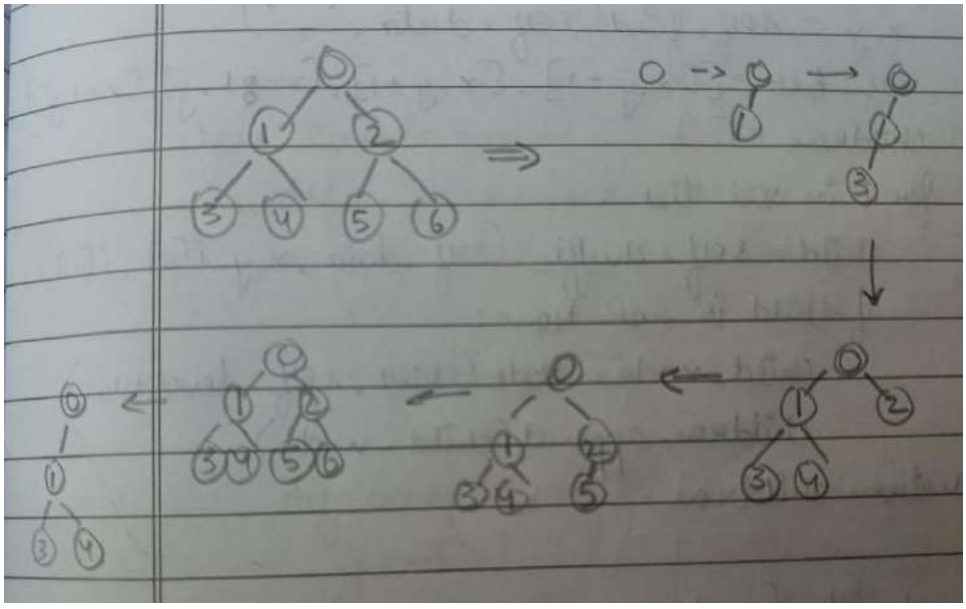
PAGE NO :
DATE :

Enter the source node : 0
Enter the Target node : 7
Enter the maximum depth : 4

Target 7 is reachable from source 0 within max depth 4

Traversal path : 0 → 1 → 3 → 7

18/12/23



2.5 Experiment - 5

2.5.1 Question:

Implement 8 puzzle using A* Algorithm.

2.5.2 Code:

```
class Node:
    def __init__(self,data,level,fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x,y = self.find(self.data,'_')
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

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


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

    def find(self,puz,x):
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
```

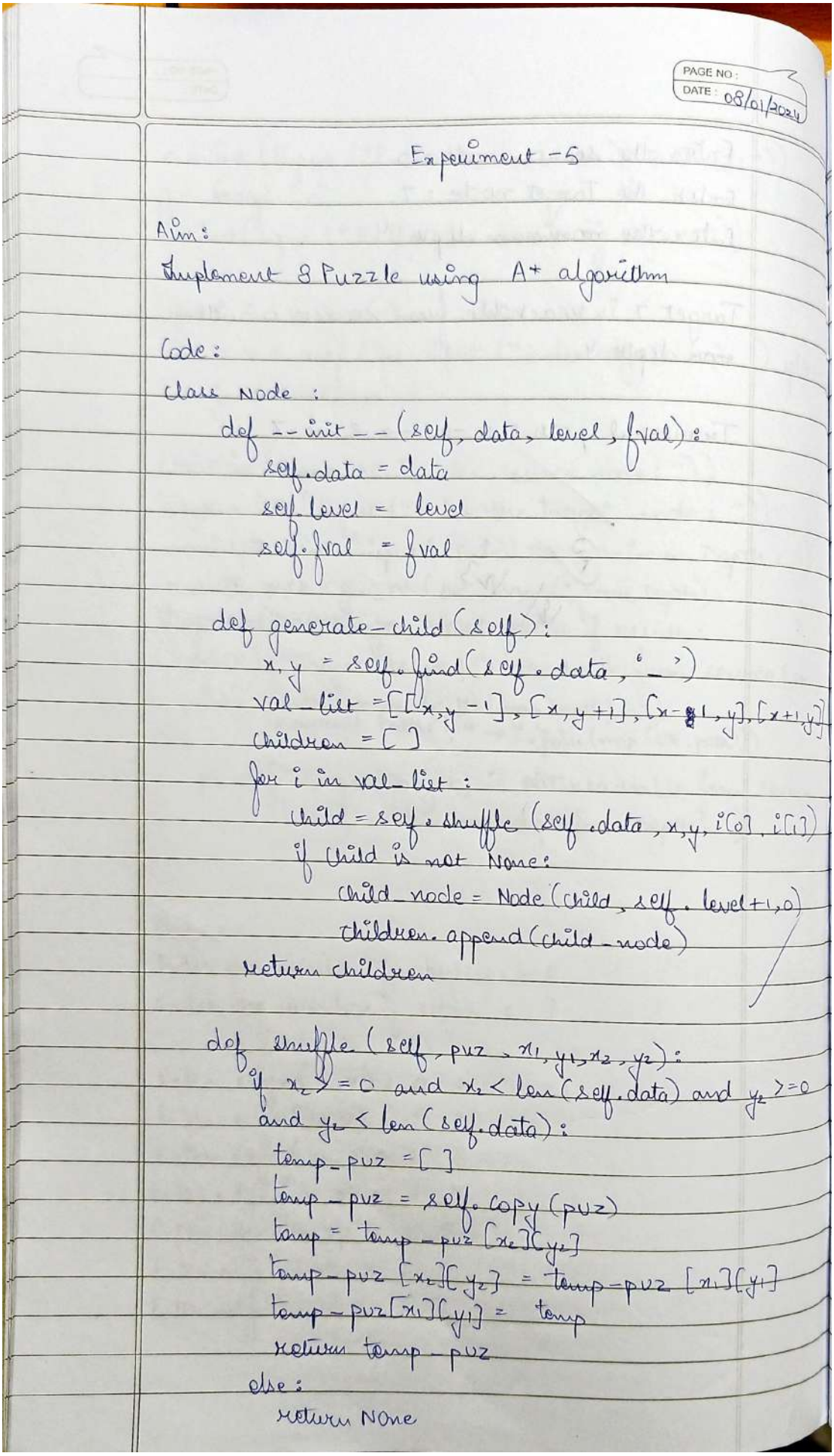
```
            if puz[i][j] == x:
                return i,j


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

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

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

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


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

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\'/ \n")
            for i in cur.data:
```

```
            for j in i:
                print(j,end=" ")
            print("")
        if(self.h(cur.data,goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i,goal)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]

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


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

2.5.3 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 _
```

2.5.4 Observation Book Pictures:



PAGE NO :
DATE : 08/01/2024

Experiment -5

Aim:

Implement 8 Puzzle using A* algorithm

Code:

```
class Node :
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '_')
        val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data, x, y, i[0], i[1])
            if child is not None:
                child_node = Node(child, self.level+1, 0)
                children.append(child_node)
        return children

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

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

def find(self, puz, x):
    for i in range(0, len(self.data)):
        for j in range(0, len(self.data)):
            if puz[i][j] == x:
                return i, j

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

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

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

```
def h(self, start, goal):
    temp = 0
    for i in range(0, self.n):
        for j in range(0, self.n):
            if start[i][j] != goal[i][j] and start[i][j] !
                temp += 1
    return temp

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

    start = Node(start, 0, 0)
    start.fval = self.f(start, goal)
    self.open.append(start)
    print("\n\n")
    while True:
        cur = self.open[0]
        print("")
        print(" | ")
        print(" | ")
        print(" \\'/ \n")
        for i in cur.data:
            for j in i:
                print(j, end=" ")
            print("")
        if(self.h(cur.data, goal) == 0):
            break
        for i in cur.generate_child():
            i.fval = self.f(i, goal)
            self.open.append(i)
```

```
        self.closed.append(cur)
    def self.open[0]

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

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

Output:

Enter start state matrix:

1 2 3
4 5 _
7 8 6

Enter goal state matrix:

1 2 3
4 5 6
7 8 _

↓

1 2 3
4 5 _
7 8 6

↓

1 2 3
4 5 6
7 8 _

2.6 Experiment - 6

2.6.1 Question:

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

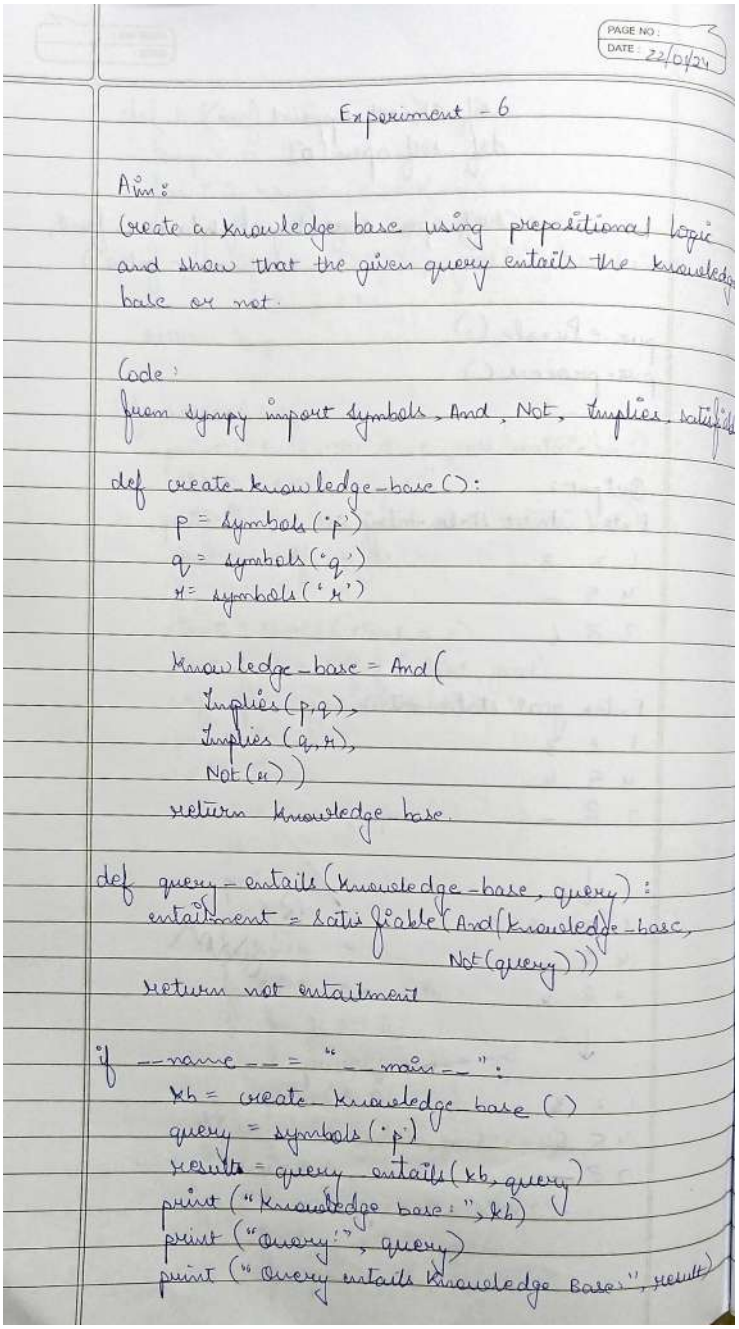
2.6.2 Code:

```
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)
```

2.6.3 Output:

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

2.6.4 Observation Book Pictures:

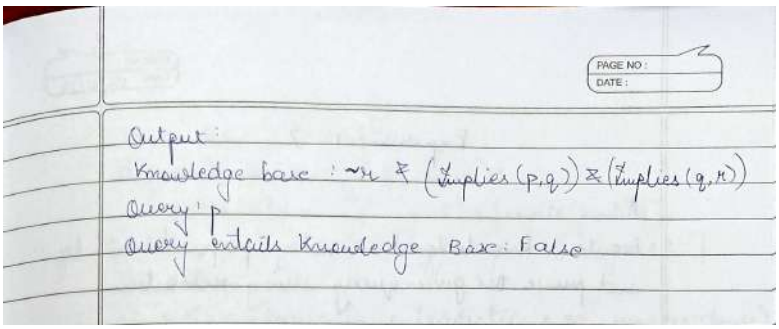


Experiment - 6

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

Code:

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

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

    knowledge_base = And(
        Implies(p,q),
        Implies(q,r),
        Not(r))
    return knowledge_base

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

if __name__ == "__main__":
    kb = create_knowledge_base()
    query = symbols('p')
    result = query_entails(kb, query)
    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

2.7 Experiment - 7

2.7.1 Question:

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

2.7.2 Code:

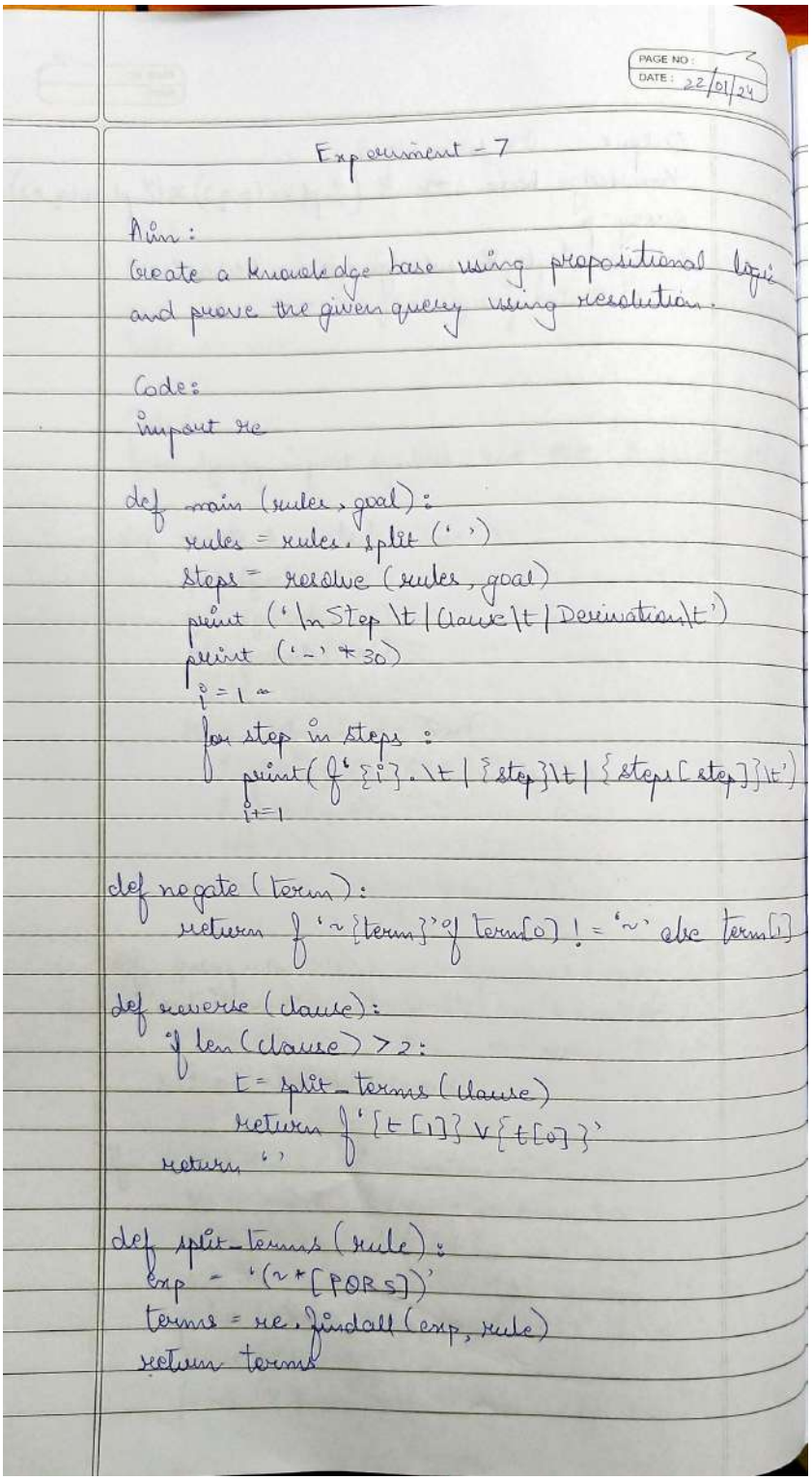
```
import re

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)
```

2.7.3 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.
```

2.7.4 Observation Book Pictures:



Experiment - 7

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

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]}'):
```

PAGE NO :
DATE :

temp.append(f'{gen[0]} v {gen[1]}')
steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \n A contradiction is found when {negate(goal)} is assumed 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. \n A contradiction is found when {negate(goal)} is assumed as true. Hence, {goal} is true."
return steps

for clause in clauses:
if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:
temp.append(clause)
steps[clause] = f"Resolved from {temp[i]} and {temp[j]}."
j = (j+1)%n
i += 1
return steps

PAGE NO :
DATE :

rules = 'R v ~P R v ~Q ~R v P ~R v Q ~R v Q'
goal = 'R'
main (rules, goal)

Output

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

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

2.8 Experiment - 8

2.8.1 Question:

Implement unification in first order logic.

2.8.2 Code:

```
import re

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

def 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)
```

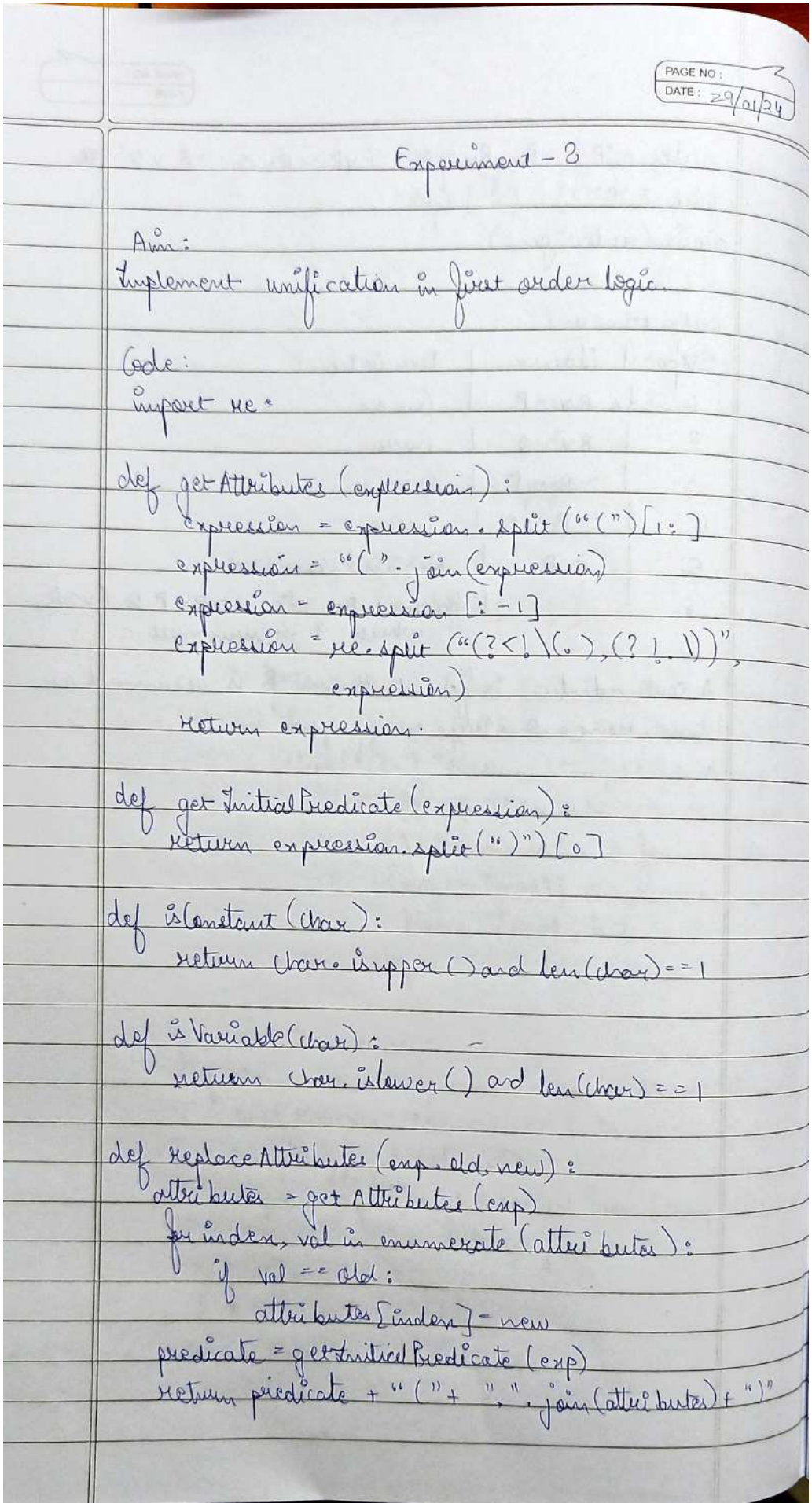
```
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression
def unify(exp1, exp2):
    if exp1 == exp2:
        return []

    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)
```

2.8.3 Output:

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

2.8.4 Observation Book Pictures:



PAGE NO :
DATE : 29/01/24

Experiment - 3

Aim:
Implement unification in first order logic.

Code:

```
import re

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

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

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

def unify(exp1, exp2):
    if exp1 == exp2
        if exp1 != 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")

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)

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

initialSubstitutions.extend(remainingSubstitutions)
return initialSubstitutions

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

Output:

Substitutions:

[('X', 'Richard')]

2.9 Experiment - 9

2.9.1 Question:

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

2.9.2 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))
```

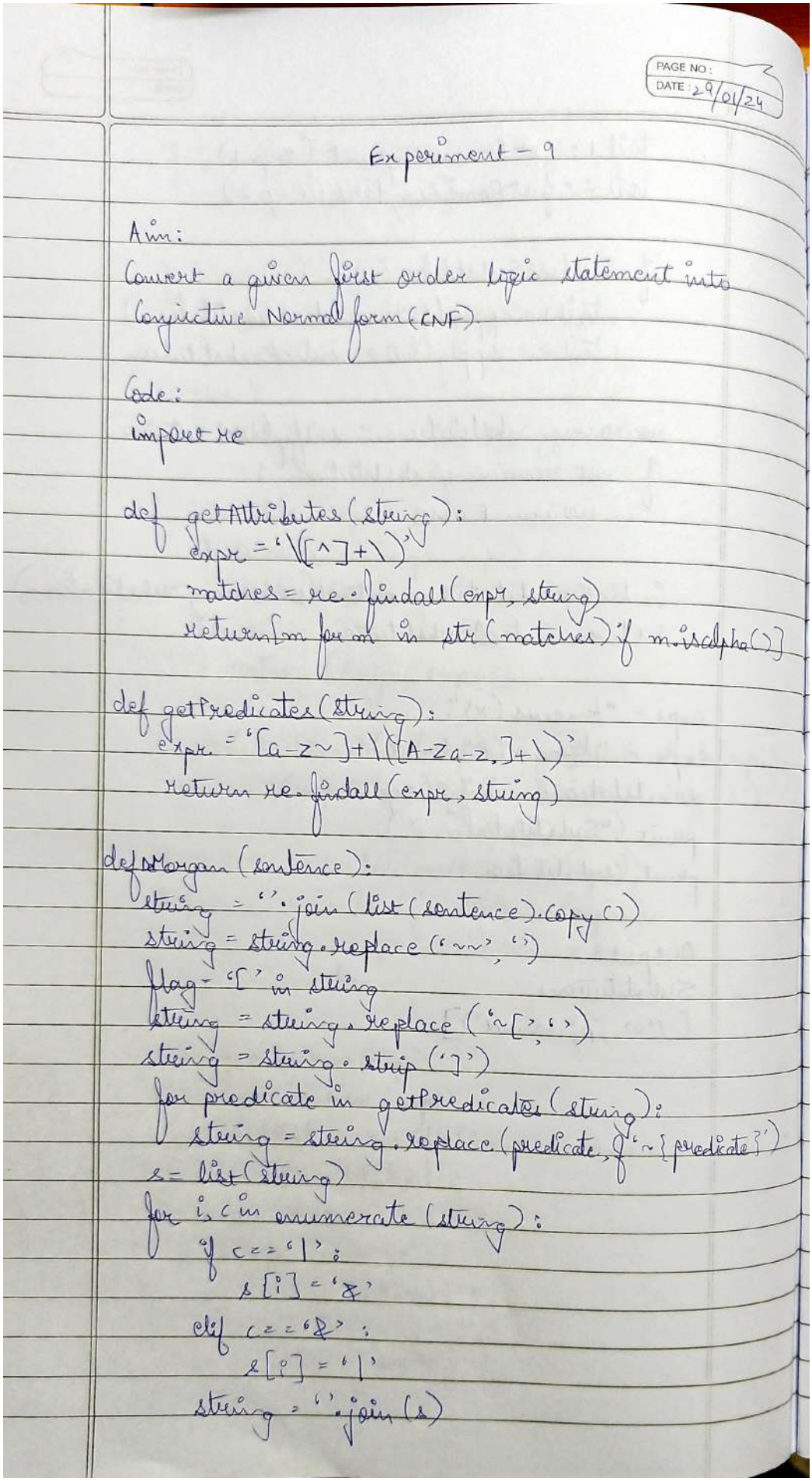
```
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement

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

2.9.3 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)
```

2.9.4 Observation Book Pictures:



PAGE NO :
DATE : 29/01/24

Experiment - 9

Aim:
Convert a given first order logic statement into Conjuctive Normal form (CNF).

Code:

```
import re

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('~~', '')
        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]:
        statements = 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

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 = '(~[∀|E].)'
    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) => loved(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(F(x)) & ~loved(x,F(x))] | [loves(F(x),x)]
[~american(x) | ~weapon(y) | ~sells(x,y,z) | ~hostile(z)] | criminal(x)
```

2.10 Experiment - 10

2.10.1 Question:

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

2.10.2 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)
```

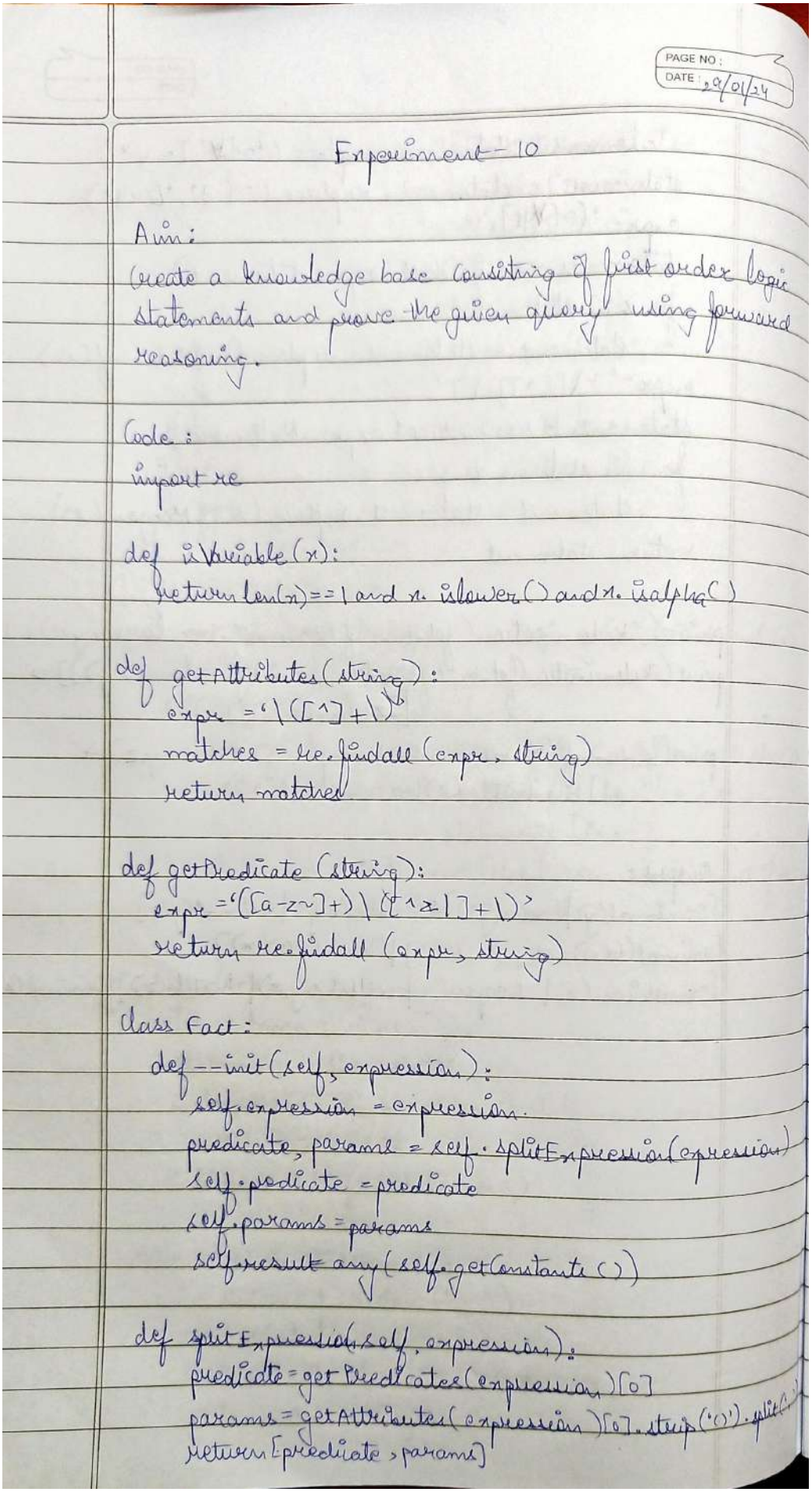
```
class Implication:
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    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()
```

2.10.3 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)
```

2.10.4 Observation Book Pictures:



PAGE NO :
DATE : 9/01/24

Experiment - 10

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

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 getPredicate(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),
                        str(getAttributes(self.rhs.expression))
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):
    criminal(west)

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

3. Certificates of Python:

3.1 Kaggle:

3.2 Infosys Springboard: