VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
On

ARTIFICIAL INTELLIGENCE LABORATORY

Submitted by

P SAI KRISHNA (1BM21CS123)

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

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

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

CERTIFICATE

This is to certify that the Lab work entitled **“ARTIFICIAL INTELLIGENCE LABORATORY”** carried out by **P SAI KRISHNA (1BM21CS123)**, who is bonafide student of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2022-23. The Lab report has been approved as it satisfies the academic requirements in respect of **Artificial Intelligence Lab - (22CS5PCAIN)** work prescribed for the said degree.

Saritha AN
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

Table of Contents

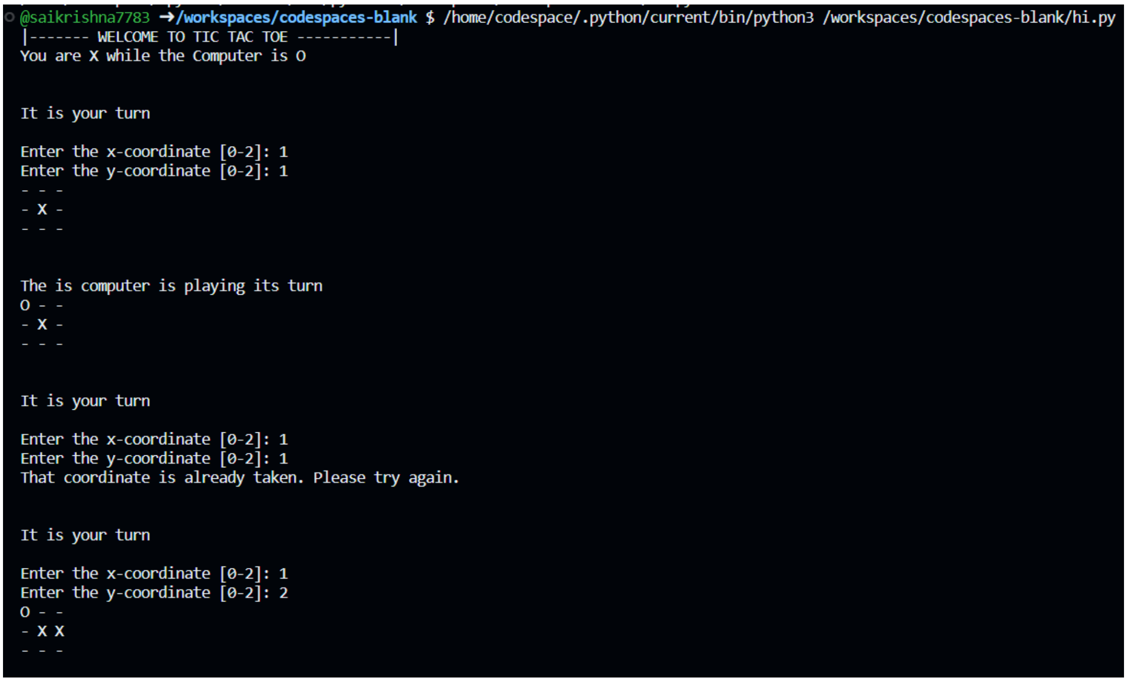
---

1. Implement Tic –Tac –Toe Game.

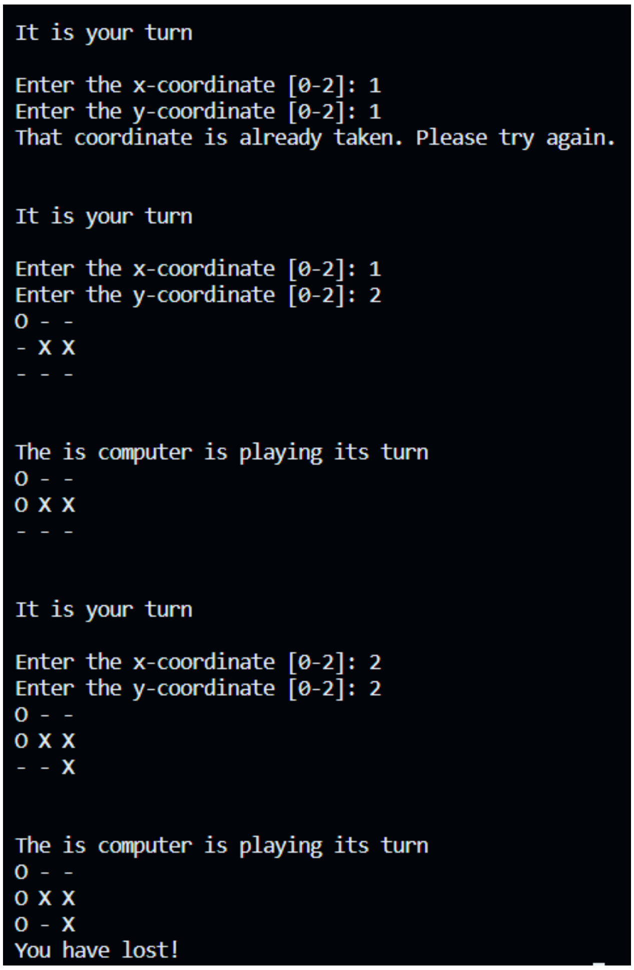
Program:

```
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")
```

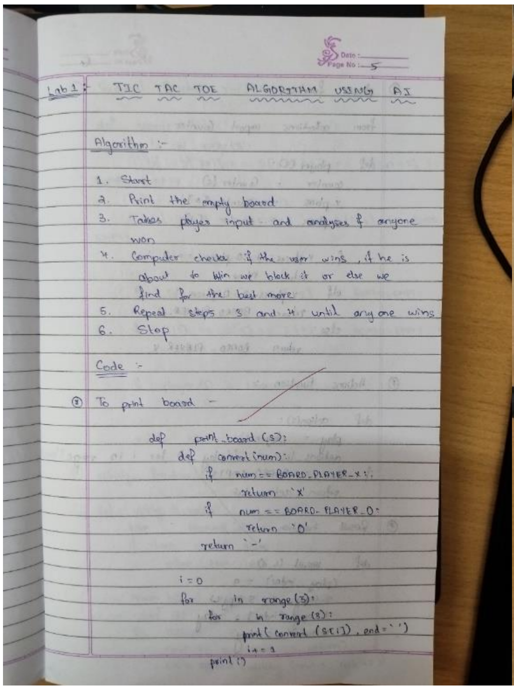
Output :



```
○ @saikrishna7783 ➜ /workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/hi.py
 |------- WELCOME TO TIC TAC TOE -----------|
 You are X while the Computer is O


 It is your turn

 Enter the x-coordinate [0-2]: 1
 Enter the y-coordinate [0-2]: 1
 - - -
 - X -
 - - -


 The is computer is playing its turn
 O - -
 - X -
 - - -


 It is your turn

 Enter the x-coordinate [0-2]: 1
 Enter the y-coordinate [0-2]: 1
 That coordinate is already taken. Please try again.


 It is your turn

 Enter the x-coordinate [0-2]: 1
 Enter the y-coordinate [0-2]: 2
 O - -
 - X X
 - - -
```



```
It is your turn

Enter the x-coordinate [0-2]: 1
Enter the y-coordinate [0-2]: 1
That coordinate is already taken. Please try again.


It is your turn

Enter the x-coordinate [0-2]: 1
Enter the y-coordinate [0-2]: 2
O - -
- X X
- - -


The is computer is playing its turn
O - -
O X X
- - -


It is your turn

Enter the x-coordinate [0-2]: 2
Enter the y-coordinate [0-2]: 2
O - -
O X X
- - X


The is computer is playing its turn
O - -
O X X
O - X
You have lost!
```



Date :
Page No : 5

Lab 1 : TIC TAC TOE ALGORITHM USING AI

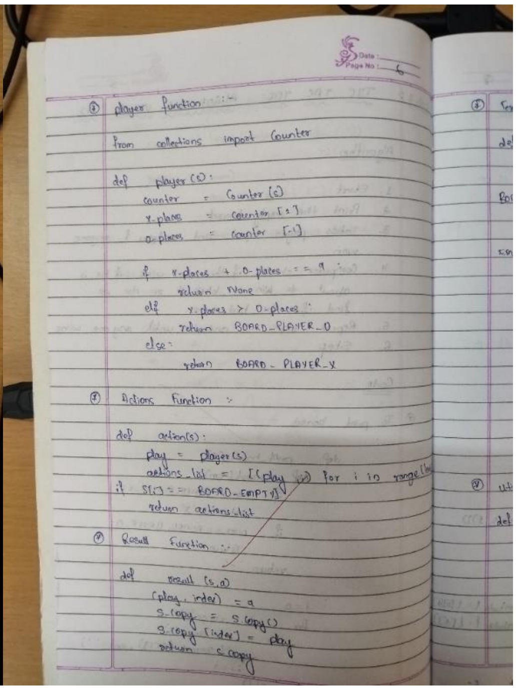
Algorithm :-

1. Start
2. Print the empty board
3. Takes player input and analyses if anyone won
4. Computer checks if the user wins, if he is about to win we block it or else we find for the best move
5. Repeat steps 3 and 4 until any one wins
6. Stop

Code :-

① To print board -

```
def  print_board (s):
    def  convert(num):
        if  num == BOARD_PLAYER_X :
            return 'X'
        if  num == BOARD_PLAYER_O :
            return 'O'
    return '-'

    i = 0
    for   in  range(3):
        for   in  range (3):
            print( convert (s[i]), end=' ')
            i+ = 1
    print ()
```



Date :
Page No : 6

② player function :-

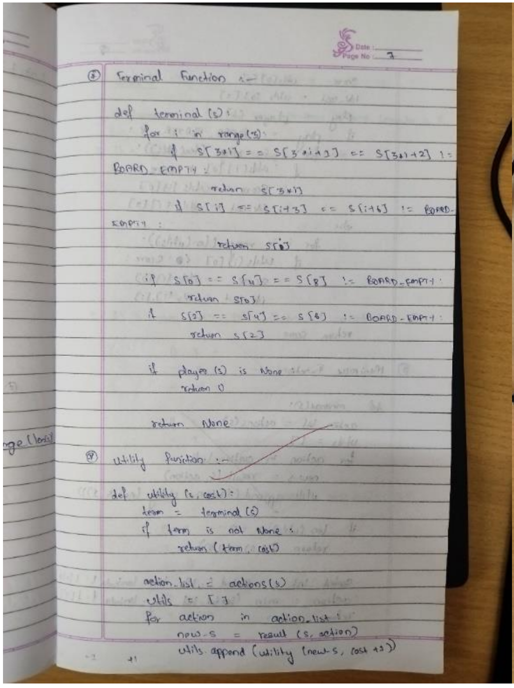
```
from  collections  import  Counter

def  player (s):
    counter  =  Counter (s)
    x_places  =  counter [1]
    o_places  =  counter [-1]

    if  x_places + o_places == 9 :
        return  None
    elif  x_places > o_places :
        return  BOARD_PLAYER_O
    else:
        return  BOARD_PLAYER_X
```

③ Actions Function :

```
def  action(s):
    play = player(s)
    actions_list = [(play, i) for i in range(len
if  s[i] == BOARD_EMPTY]
    return  actions_list
```

④ Result Function :-

```
def  result (s, a)
    (play, index) = a
    s_copy = s.copy()
    s_copy [index] = play
    return  s_copy
```



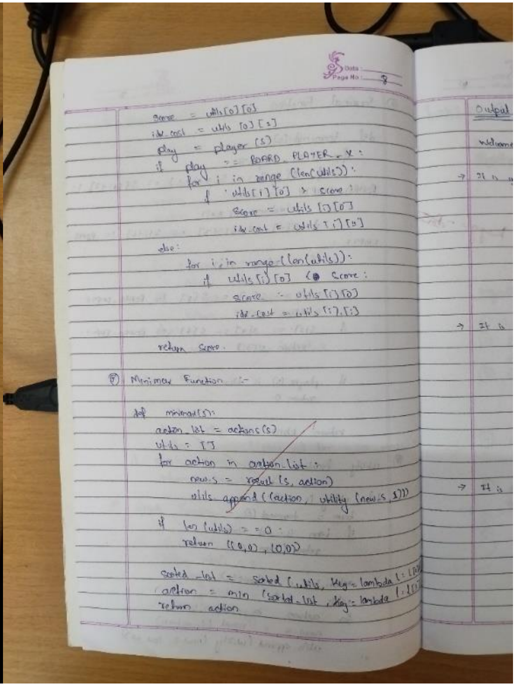
Date :
Page No : 7

⑤ Terminal Function :-

```
def  terminal (s):
    for  i  in  range(3):
        if  s[3*i] == s[3*i+1] == s[3*i+2] !=
BOARD_EMPTY :
            return  s[3*i]
        if  s[i] == s[i+3] == s[i+6] != BOARD-
EMPTY :
            return  s[i]

    if  s[0] == s[4] == s[8] != BOARD_EMPTY :
        return  s[0]
    if  s[2] == s[4] == s[6] != BOARD_EMPTY :
        return  s[2]

    if  player (s)  is  None
        return 0

    return  None
```

⑥ utility function :-

```
def  utility (s, cost):
    term  =  terminal (s)
    if  term  is  not  None :
        return ( term , cost)

    action_list  =  actions(s)
    utils  =  [ ]
    for  action  in  action_list :
        new_s  =  result (s, action)
        utils append (utility (new_s, cost +1))
```



Date :
Page No : 8

```
    score = utils[0][0]
    idx_cost = utils [0] [1]
    play = player (s)
    if  play == BOARD_PLAYER_X :
        for  i  in  range (len(utils)):
            if  utils[i] [0] > score :
                score = utils [i] [0]
                idx_cost = utils [i] [1]
    else:
        for i, in range (len(utils)):
            if  utils [i] [0] < score :
                score = utils [i] [0]
                idx_cost = utils [i] [1]

    return  score.
```

⑦ Minimax Function :-

```
def  minimax(s):
    action_list = actions(s)
    utils = []
    for action in action_list :
        new_s = result (s, action)
        utils append ((action, utility (new_s, 1)))

    if  len (utils) == 0 :
        return ((0,0), (0,0))

    sorted_list = sorted (utils, key= lambda l : l[0
    action = min (sorted_list, key= lambda l : l[1
    return  action
```

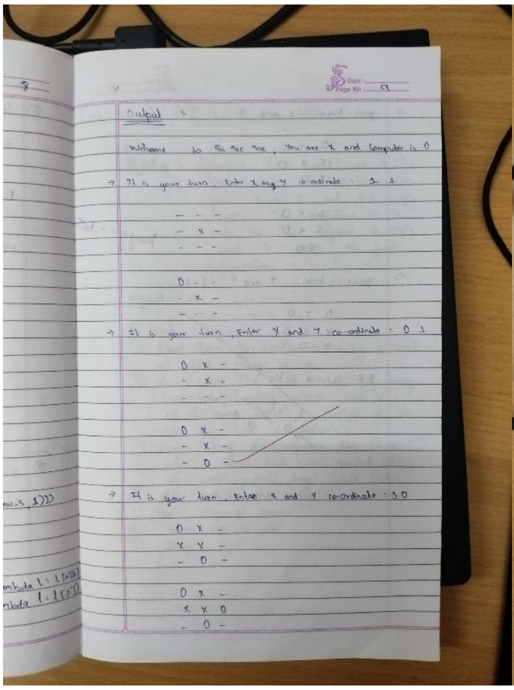
Output :

welcome

→ It is

→ It is

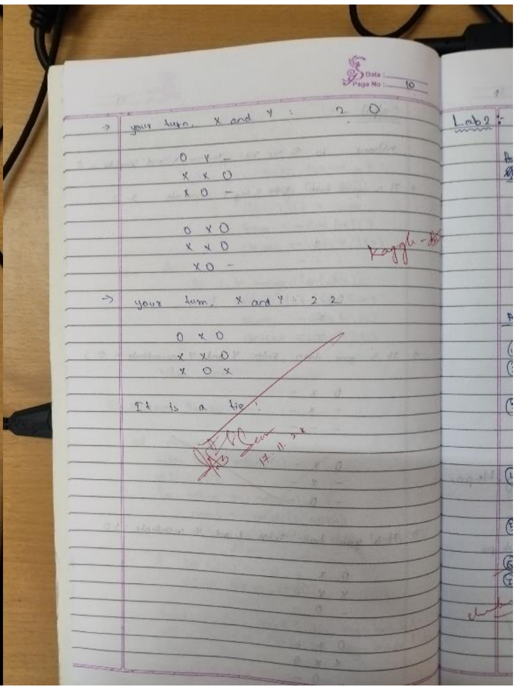
→ It is



Date :
Page No : 9

Output :

```
Welcome to Tic Tac Toe, You are X and Computer is 0

-> It is your turn, Enter X and Y co-ordinate : 1 1

   - - -
   - X -
   - - -

   0 - -
   - X -
   - - -

-> It is your turn, Enter X and Y co-ordinate : 0 1

   0 X -
   - X -
   - - -

   0 X -
   - X -
   - 0 -

-> It is your turn, Enter X and Y co-ordinate : 1 0

   0 X -
   X X -
   - 0 -

   0 X -
   X X 0
   - 0 -
```



Date :
Page No : 10

```
-> your turn, X and Y : 2 0

   0 X -
   X X 0
   X 0 -

   0 X 0
   X X 0
   X 0 -

-> your turn, X and Y : 2 2

   0 X 0
   X X 0
   X 0 X

It is a tie !
```

Kaggle

17.11.24

Lab2 :

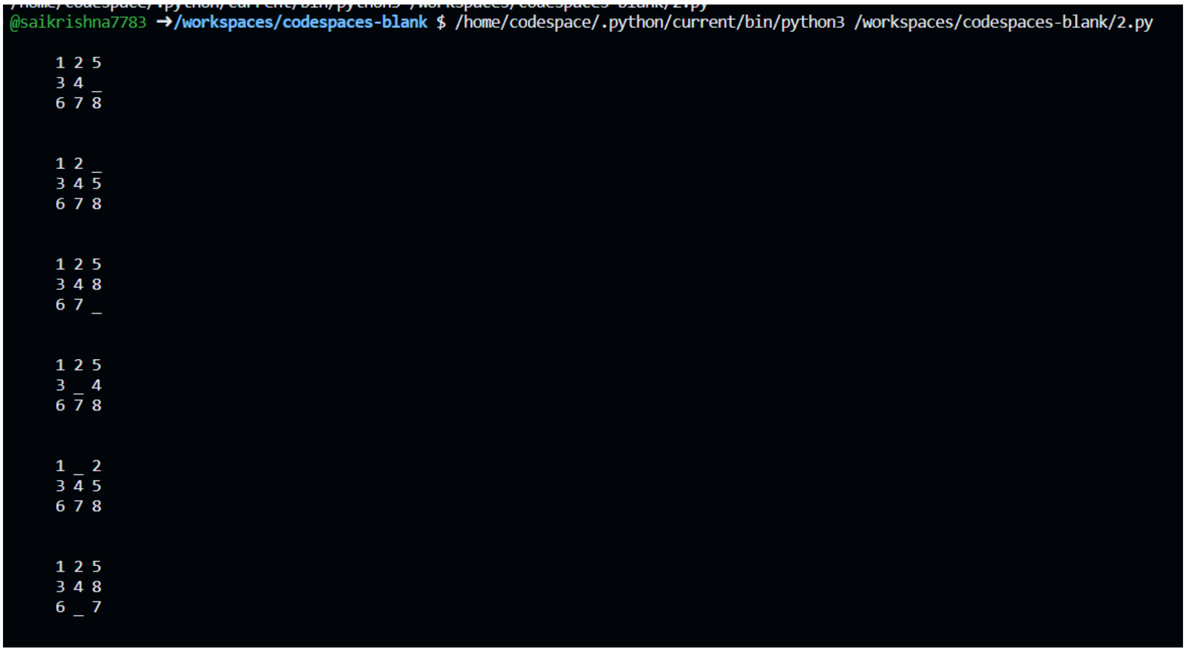
2. Solve 8 puzzle problems

Program :

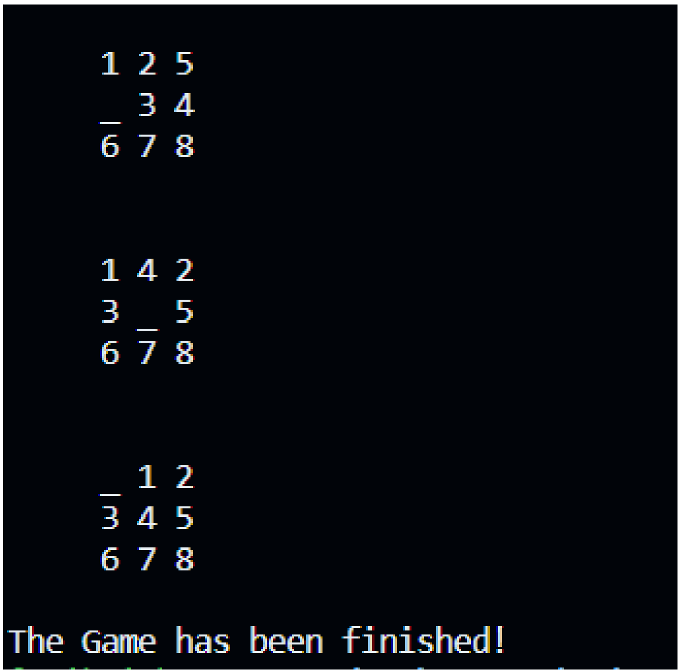
```
def bfs(src,target):
    queue=[]
    queue.append(src)
    exp=[]
    while len(queue)>0:
        source=queue.pop(0)
        #print("queue",queue)
        exp.append(source)

        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)
```

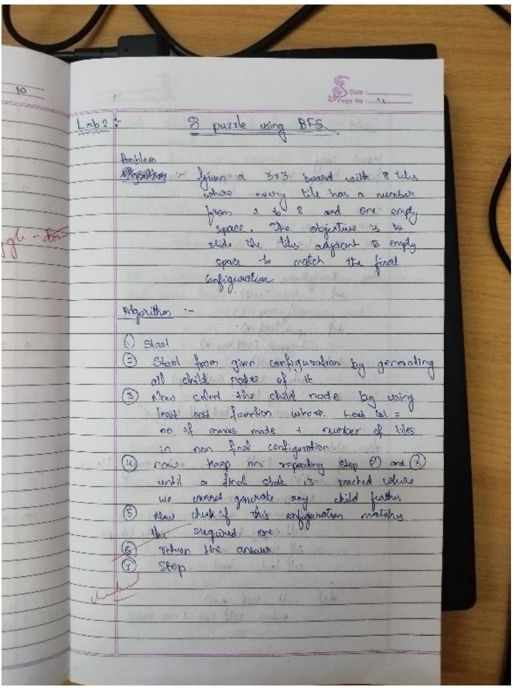
Output:



```
@saikrishna7783 →/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/2.py

    1 2 5
    3 4 _
    6 7 8


    1 2 _
    3 4 5
    6 7 8


    1 2 5
    3 4 8
    6 7 _


    1 2 5
    3 _ 4
    6 7 8


    1 _ 2
    3 4 5
    6 7 8


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



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


    1 4 2
    3 _ 5
    6 7 8


    _ 1 2
    3 4 5
    6 7 8

The Game has been finished!
```



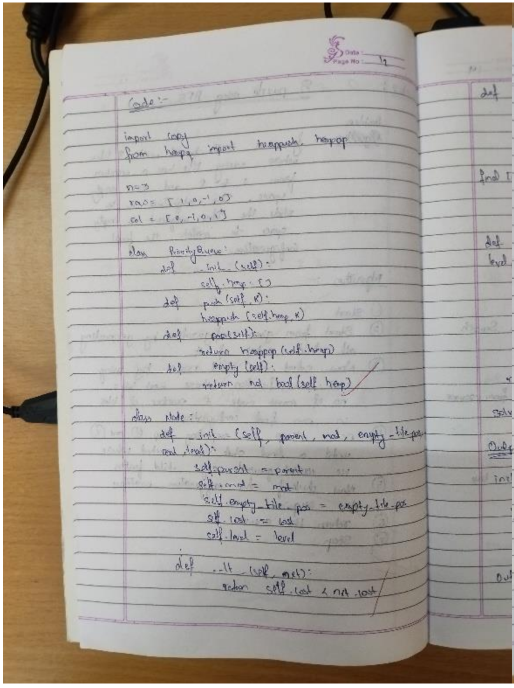
Page No : 11

Lab2 :- 8 puzzle using BFS

Problem ~~Algorithm~~ :- given a 3×3 board with 8 tiles where every tile has a number from 1 to 8 and one empty space. The objective is to slide the tiles adjacent to empty space to match the final configuration

Algorithm :-

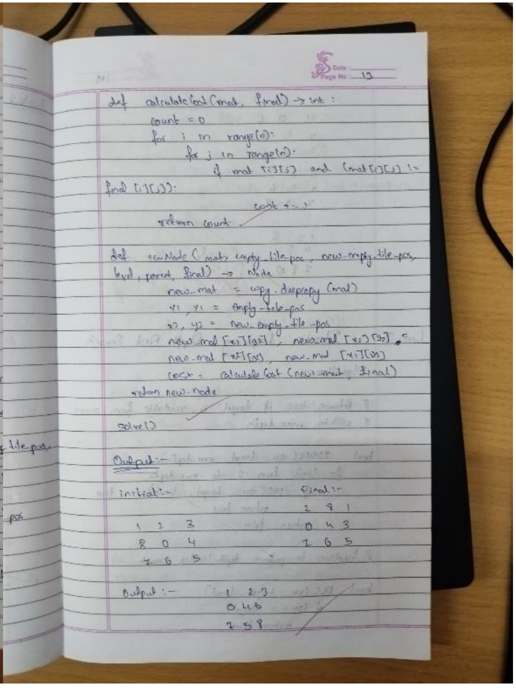
```
(1) Start
(2) Start from given configuration by generating
    all child nodes of it.
(3) Now select the child node by using
    least cost function where. Least (c) =
    no of moves made + number of tiles
    in non final configuration.
(4) now keep on repeating step (2) and (3)
    until a final state is reached where
    we cannot generate any child further
(5) Now check if this configuration matches
    the required one.
(6) return the answer
(7) Stop.
```



Page No : 12

Code :-

```
import copy
from heapq import heappush, heappop

n=3
row = [1,0,-1,0]
col = [0,-1,0,1]

class PriorityQueue:
      def __init__(self):
            self.heap = []
      def push(self, k):
            heappush (self.heap, k)
      def pop(self):
            return heappop (self.heap)
      def empty (self):
            return not bool(self heap)

class Node :
      def __init__(self, parent, mat, empty_tile_pos,
      cost, level):
            self.parent = parent
            self.mat = mat
            self.empty_tile_pos = empty_tile_pos
            self.cost = cost
            self.level = level

      def __lt__(self, nxt):
            return self.cost < nxt.cost
```

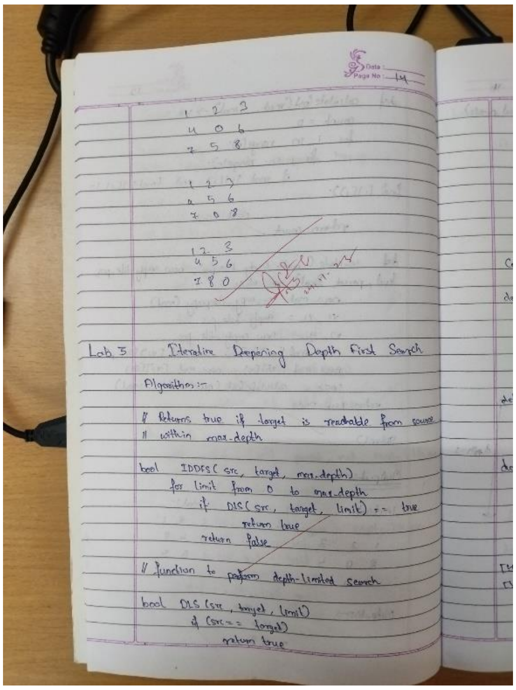


Page No : 13

```
def calculateCost(mat, final) -> int :
      count = 0
      for i in range(n):
            for j in range(n):
                  if mat [i][j] and (mat[i][j] !=
final [i][j]):
                        count += 1
      return count

def newNode ( mat, empty_tile_pos, new_empty_tile_pos,
level, parent, final) -> Node
            new_mat = copy.deepcopy (mat)
            x1, y1 = empty_tile_pos
            x2, y2 = new_empty_tile_pos
            new_mat [x1][y1], new_mat [x2][y2] =
            new_mat [x2][y2], new_mat [x1][y1]
            cost = calculateCost (new_mat, final)
      return new_node

solve()
```

Output :-

```
initial :-                  Final :-
                            2 8 1
  1  2  3                   0 4 3
  8  0  4                   7 6 5
  7  6  5

Output :-        1 2 3
                 0 4 6
                 7 5 8
```



Page No : 14

```
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
```

Lab 5 Iterative Deepening Depth First Search

Algorithm :-

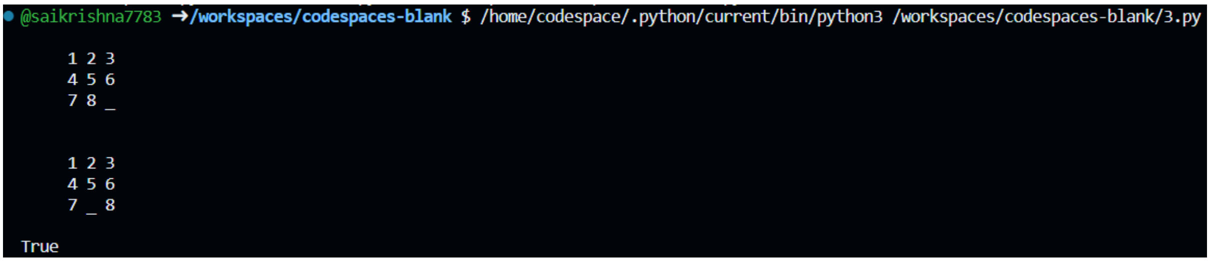
```
// Returns true if target is reachable from source
// within max-depth

bool IDDFS( src, target, max-depth)
    for limit from 0 to max-depth
        if DLS( src, target, limit) == true
            return true
        return false

// function to perform depth-limited search

bool DLS (src, target, limit)
    if (src == target)
        return true
```

3. Implement Iterative deepening search algorithm.

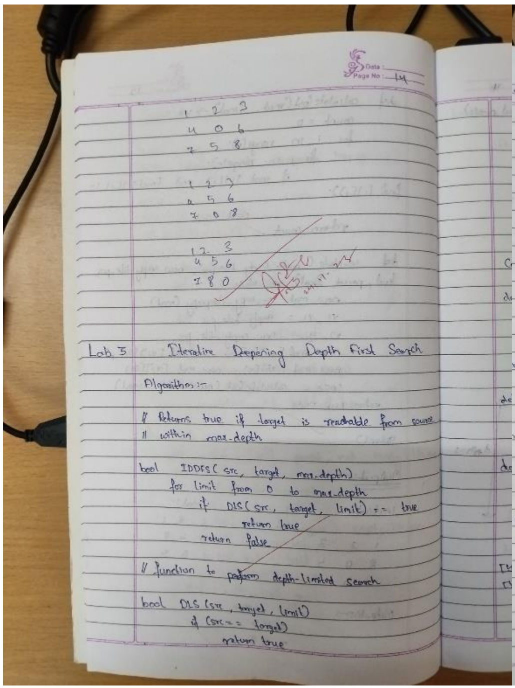
Program:

```
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:



```
● @saikrishna7783 ➜/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/3.py

     1 2 3
     4 5 6
     7 8 _


     1 2 3
     4 5 6
     7 _ 8

 True
```



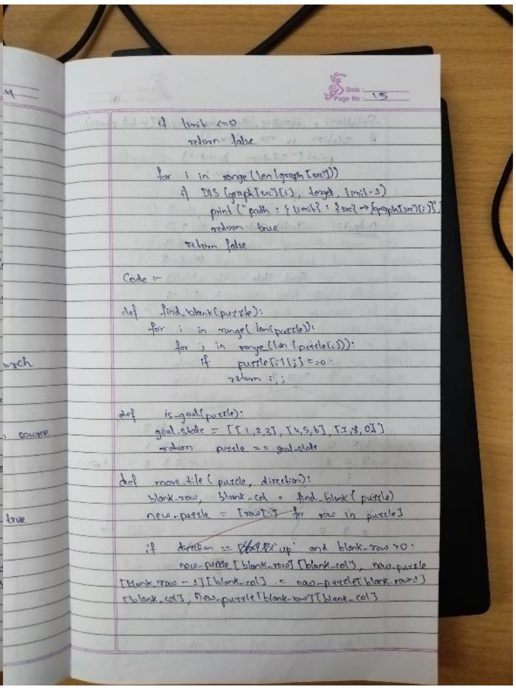
Page No : 14

```
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
```

Lab 5 Iterative Deepening Depth First Search

Algorithm :-

```
// Returns true if target is reachable from source
// within max-depth

bool  IDDFS( src, target, max-depth)
    for limit from 0 to max-depth
        if DLS( src, target, limit) == true
            return true
        return false

// function to perform depth-limited search

bool DLS (src, target, limit)
    if (src == target)
        return true
```

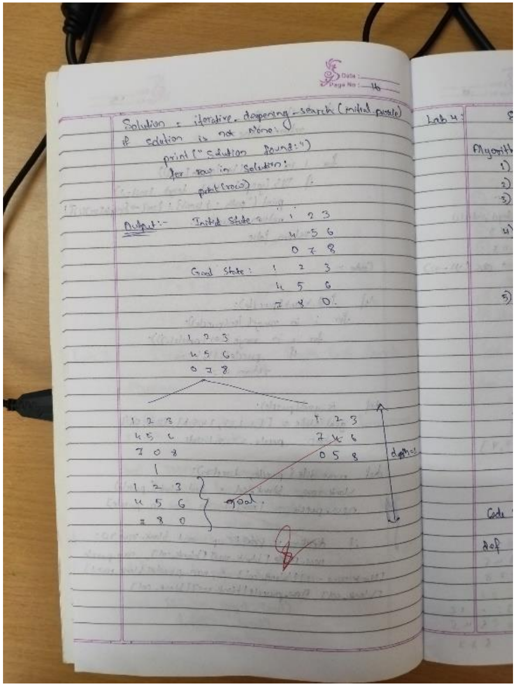


Page No : 15

```
    if limit <= 0
        return false

    for i in range(len(graph[src]))
        if DLS (graph[src][i], target, limit-1)
            print ("path : {limit} : {src} -> {graph[src][i]}")
            return true
        return false
```

Code :-

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

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

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

    if  direction == 'up' and blank_row > 0 :
            new_puzzle [blank_row] [blank_col], new_puzzle
[blank_row - 1] [blank_col] = new_puzzle[blank_row-1]
[blank_col], new_puzzle[blank_row][blank_col]
```



Page No : 16

```
Solution = iterative_deepening_search (initial_puzzle)
if solution is not None:
    print ("Solution found:")
    for row in solution:
        print(row)
```

Output :-

```
Initial State :   1 2 3
                  4 5 6
                  0 7 8

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

          1 2 3
          4 5 6
          0 7 8

1 2 3                      1 2 3
4 5 6                      7 4 6
7 0 8                      0 5 8     depth=1
  |
1 2 3
4 5 6  }  goal.
7 8 0
```

4. Implement A* search algorithm.

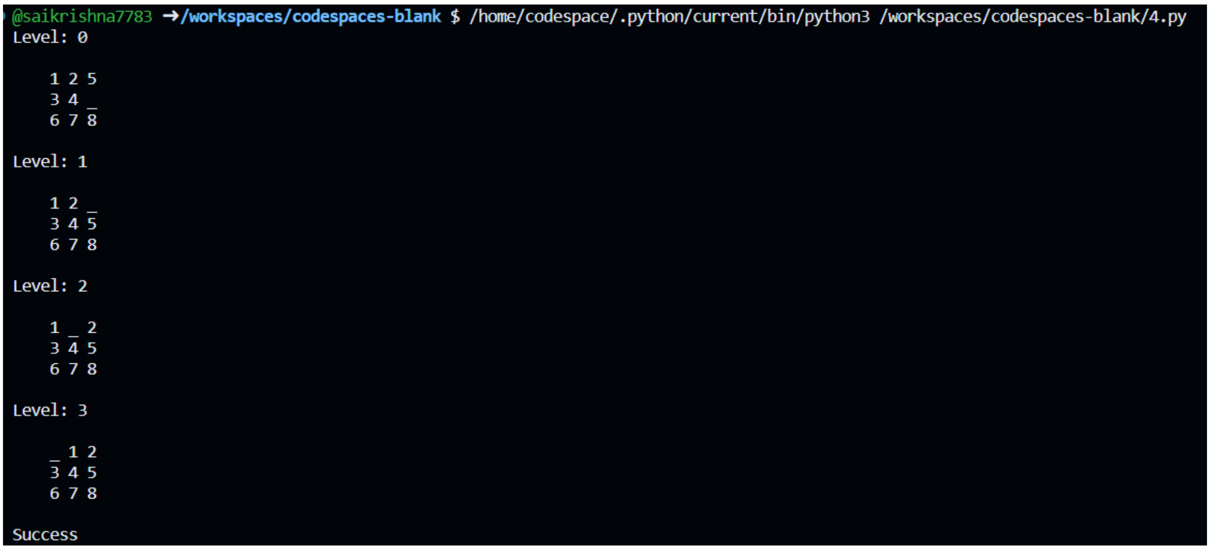
Program:

class Node:

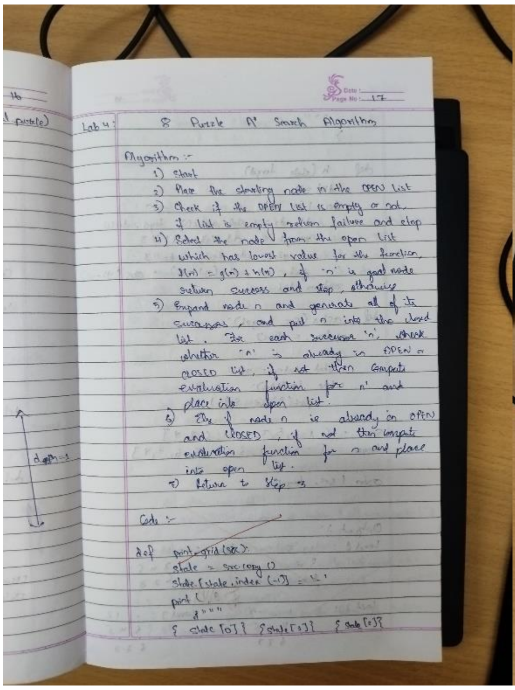
```
def __init__(self,data,level,fval):
    """ Initialize the node with the data, level of the node and the calculated fvalue """
    self.data = data
    self.level = level
    self.fval = fval

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

def shuffle(self,puz,x1,y1,x2,y2):
    """ Move the blank space in the given direction and if the position value are out
        of limits the return None """
    if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
        temp_puz = []
        temp_puz = self.copy(puz)
        temp = temp_puz[x2][y2]
        temp_puz[x2][y2] = temp_puz[x1][y1]
        temp_puz[x1][y1] = temp
```

```
            return temp_puz
        else:
            return None

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

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

class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

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

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

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

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

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

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



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

Output:



```
@saikrishna7783 ➜/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/4.py
Level: 0

    1 2 5
    3 4 _
    6 7 8

Level: 1

    1 2 _
    3 4 5
    6 7 8

Level: 2

    1 _ 2
    3 4 5
    6 7 8

Level: 3

    _ 1 2
    3 4 5
    6 7 8

Success
```



Page No: 17

Lab 4: 8 Puzzle A* Search Algorithm

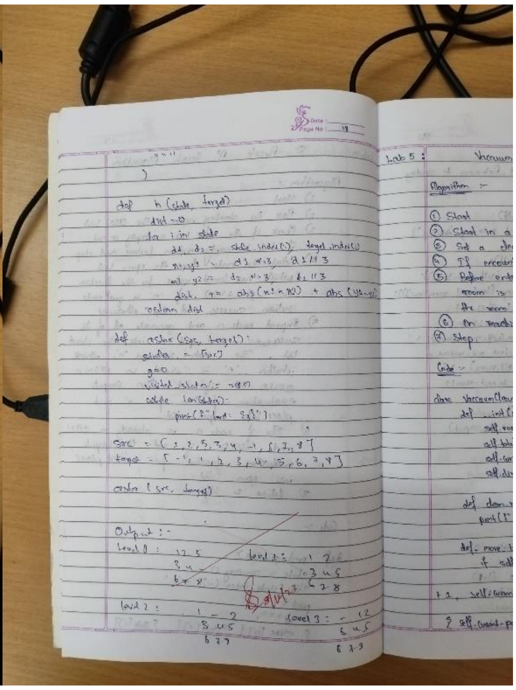
Algorithm :-

1) Start
2) Place the starting node in the OPEN list
3) Check if the OPEN list is empty or not. if list is empty return failure and stop
4) Select the node from the open list which has lowest value for the function, f(n) = g(n) + h(n), if 'n' is goal node return success and stop, otherwise
5) Expand node n and generate all of its successors, and put n into the closed list. For each successor 'n'', check whether 'n'' is already in OPEN or CLOSED list, if not then compute evaluation function for n' and place into open list.
6) Else if node n is already in OPEN and CLOSED, if not then compute evaluation function for n and place into open list.
7) Return to step 3

depth = 1

Code :-

```
def print_grid(src):
    state = src.copy()
    state[state.index(-1)] = '_'
    print(
        f"""
    {state[0]} {state[1]} {state[2]}
```



Page No: 19

```
"""
    )

def h(state, target):
    dist = 0
    for i in state:
        d1, d2 = state.index(i), target.index(i)
        x1, y1 = d1 % 3, d1 // 3
        x2, y2 = d2 % 3, d2 // 3
        dist += abs(x1 - x2) + abs(y1 - y2)
    return dist

def astar(src, target):
    states = [src]
    g = 0
    visited_states = set()
    while len(states):
        print(f"Level: {g}")

src = [1, 2, 5, 3, 4, -1, 6, 7, 8]
target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]

astar(src, target)
```

Output :-

```
Level 0 : 1 2 5        Level 1 : 1 2
          3 4                    3 4 5
          6 7 8                  6 7 8

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

Lab 5 : Vacuum

Algorithm :-

① Start
② Start in a
③ Set a
④ If
⑤ Before
⑥ On
⑦ Stop.

Code :-

class VacuumClea

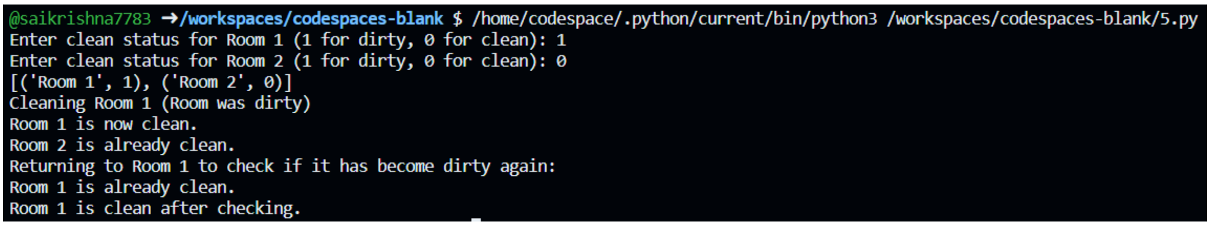
5. Implement vaccum cleaner agent.

```
def vacuum_world():
    # 0 indicates Clean and 1 indicates Dirty
```

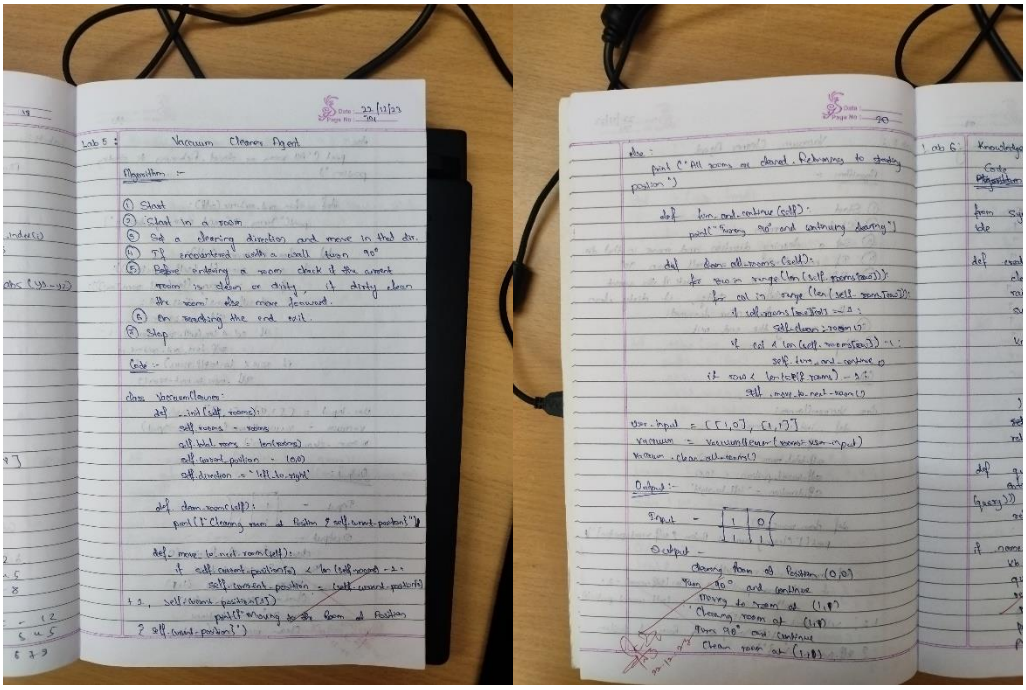
```
goal_state = {'A': '0', 'B': '0'}
cost = 0

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

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

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

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

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

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

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

# done cleaning
print("GOAL STATE: ")
```

```
    print(goal_state)

    print("Performance Measurement: " + str(cost))


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

vacuum_world()
```

OUTPUT:



```
@saikrishna7783 ➜/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/5.py
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
[('Room 1', 1), ('Room 2', 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.
```



Date: 22/12/23
Page No: 19

Lab 5 : Vaccuum Cleaner Agent

Algorithm :-

① Start
② Start in a room
③ Set a clearing direction and move in that dir.
④ If encountered with a wall turn 90°
⑤ Before entering a room check if the current room is clean or dirty, if dirty clean the room else move forward.
⑥ On reaching the end exit.
⑦ Stop

Code :-

```
class VaccumCleaner:
    def __init(self, rooms):
        self.rooms = rooms
        self.total_rooms = len(rooms)
        self.current_position = (0,0)
        self.direction = 'left_to_right'

    def clean_room(self):
        print(f"Cleaning room at Position {self.current_position}")

    def move_to_next_room(self):
        if self.current_position[0] < len(self.rooms) - 1:
            self.current_position = (self.current_position[0]
+ 1, self.current_position[1])
            print(f"Moving to the Room at Position
{self.current_position}")
```

Page No: 20

```
else:
    print("All rooms are cleaned. Returning to starting
position")

    def turn_and_continue(self):
        print("Turning 90° and continuing cleaning")

    def clean_all_rooms(self):
        for row in range(len(self.rooms[row])):
            for col in range(len(self.rooms[row])):
                if self.rooms[row][col] == 1:
                    self.clean_room()
                if col < len(self.rooms[row]) - 1:
                    self.turn_and_continue()
            if row < len(self.rooms) - 1:
                self.move_to_next_room()

user_input = [[1,0], [1,1]]
vacuum = VaccumCleaner(rooms=user_input)
vacuum.clean_all_rooms()
```

Output :-

Input - | 1 | 0 |
        | 1 | 1 |

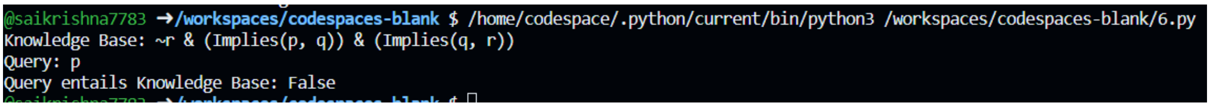
Output -
Cleaning Room at Position (0,0)
Turn 90° and continue
Moving to room at (1,0)
Cleaning room at (1,0)
Turn 90° and continue
Clean room at (1,0)

22-12-23

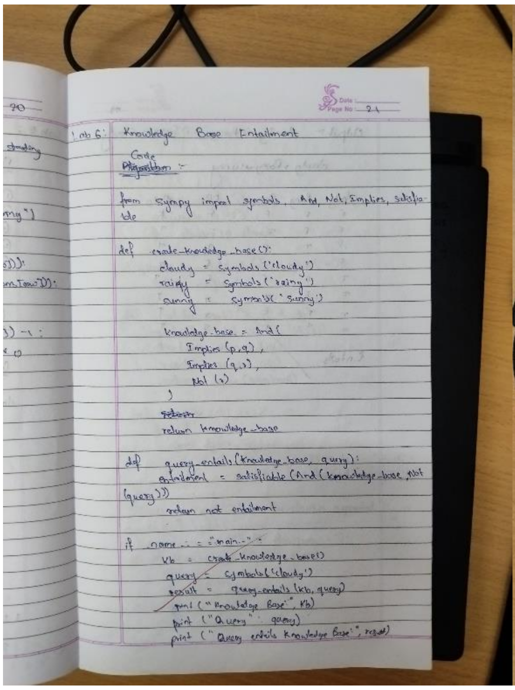
6. Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not .

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


def create_knowledge_base():

    # Define propositional symbols
    p = symbols('p')
    q = symbols('q')
    r = symbols('r')

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

    return knowledge_base

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

    # 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:



```
@saikrishna7783 →/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/6.py
Knowledge Base: ~r & (Implies(p, q)) & (Implies(q, r))
Query: p
Query entails Knowledge Base: False
```

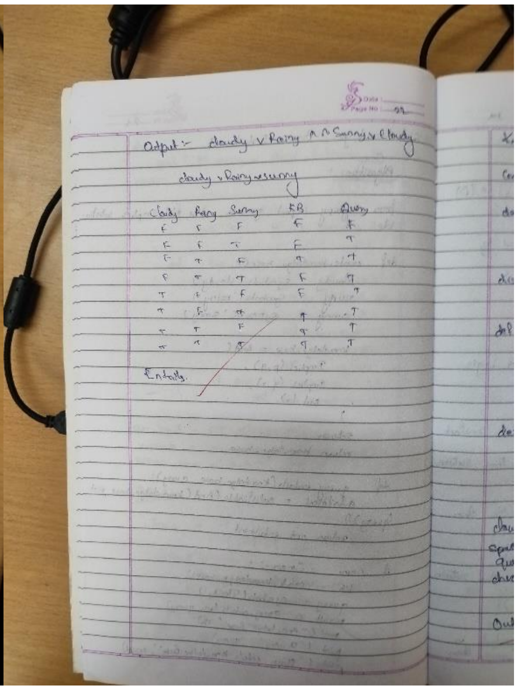


Date: Page No: 21

Lab 6: Knowledge Base Entailment

Code ~~Algorithm~~ :-

```
from sympy import symbols, And, Not, Implies, satisfia
ble

def create_knowledge_base():
    cloudy = symbols('cloudy')
    rainy = symbols('raing')
    sunny = symbols('sunny')

    knowledge_base = And(
        Implies (p,q),
        Implies (q,r),
        Not (r)
    )
    ~~return~~
    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('cloudy')
    result = query_entails(kb, query)
    print ("Knowledge Base:", kb)
    print ("Query": query)
    print ("Query entails Knowledge Base:", result)
```



Date: Page No: 22

Output:- cloudy v Rainy ∧ ¬ Sunny v Cloudy

cloudy v Rainy ~ sunny

Cloudy	Rainy	Sunny	KB	Query
F	F	F	F	F
F	F	T	F	T
F	T	F	T	T
F	T	T	F	T
T	F	F	F	T
T	F	T	T	T
T	T	F	T	T
T	T	T	T	T

Entails.

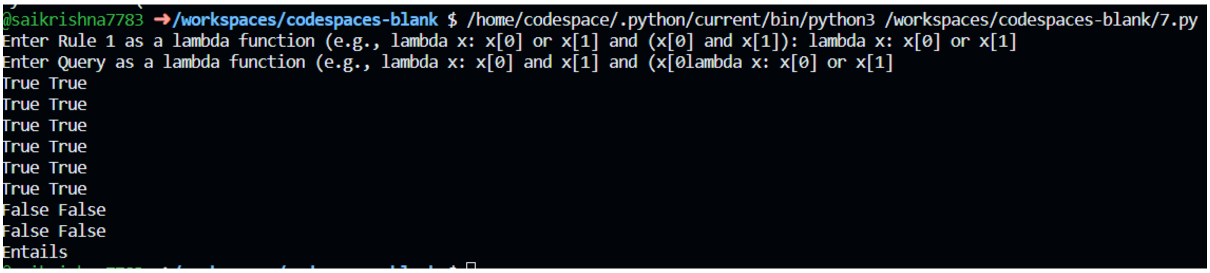
7. Create a knowledge base using prepositional logic and prove the given query using resolution

```
import re


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



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


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



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


split_terms('~PvR')
```

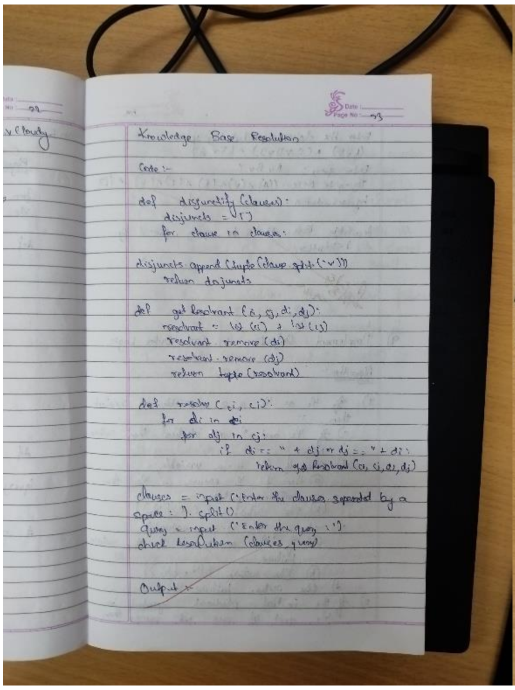
OUTPUT:

```
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)




rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR
goal = 'R'
main(rules, goal)
```



```
@saikrishna7783 ➜ /workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/7.py
Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): lambda x: x[0] or x[1]
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0lambda x: x[0] or x[1]
True True
True True
True True
True True
True True
True True
False False
False False
Entails
```



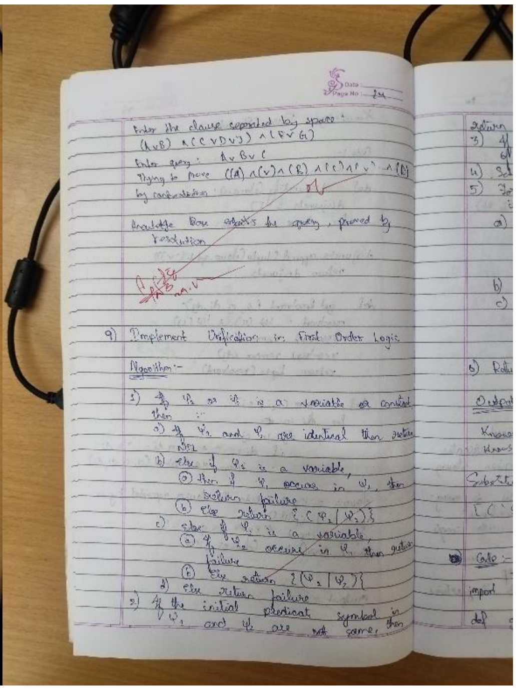
Page No: 23

Knowledge Base Resolution

Code :-

```
def disjunctify (clauses):
    disjuncts = []
    for clause in clauses:

disjuncts.append (tuple (clause.split ('v')))
    return disjuncts

def get Resolvant (ci, cj, di, dj):
    resolvant = list (ci) + list (cj)
    resolvant.remove (di)
    resolvant.remove (dj)
    return tuple (resolvant)

def resolve (ci, cj):
    for di in ci:
        for dj in cj:
            if di == '~' + dj or dj == '~' + di:
                return get Resolvant (ci, cj, di, dj)

clauses = input ('Enter the clauses separated by a
space : '). split ()
query = input ('Enter the query : ')
check Resolution (clauses, query)
```

Output :-



Page No: 24

Enter the clause separated by space :
(AvB) ∧ (CvDvJ) ∧ (FvG)
Enter query : AvBvC
Trying to prove ((A) ∧ (v) ∧ (B) ∧ (C) ∧ (v) ∧ (C)
by contradiction

Knowledge Base entails the query, proved by
resolution

9) Implement Unification in First Order Logic

Algorithm :-

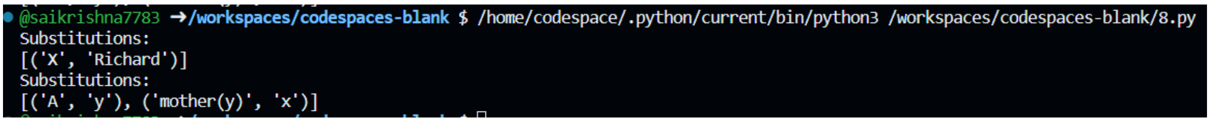
```
1) If Ψ1 or Ψ2 is a variable or constant,
   then :
   a) If Ψ1 and Ψ2 are identical then return
      NIL
   b) Else if Ψ1 is a variable,
      (a) then if Ψ1 occurs in Ψ2, then
          return failure
      (b) Else return {(Ψ2 / Ψ1)}
   c) Else if Ψ2 is a variable,
      (a) If Ψ2 occurs in Ψ1 then return
          failure
      (b) Else return {(Ψ1 / Ψ2)}
   d) Else return failure
2) If the initial predicate symbol in
   Ψ1 and Ψ2 are not same, then
```

8. Implement unification in first order logic

```
import re


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


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


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

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

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

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

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



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

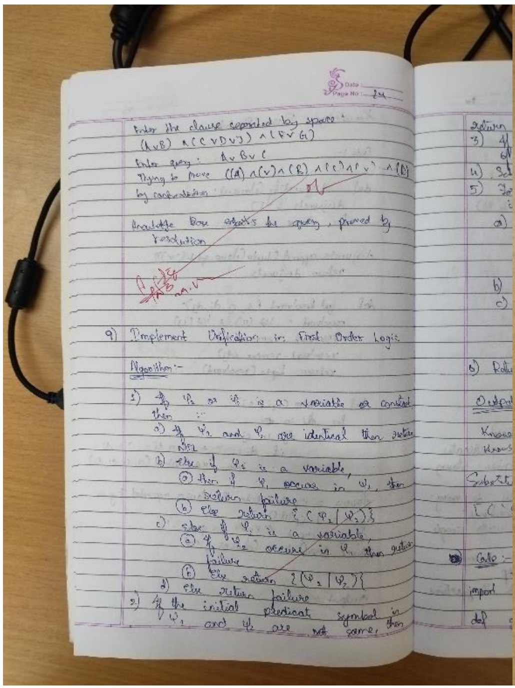
```
def getRemainingPart(expression):
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression



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

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

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

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

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

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

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

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

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

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

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

    return initialSubstitution



exp1 = "knows(X)"

exp2 = "knows(Richard)"

substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)
```

OUTPUT



```
@saikrishna7783 →/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/8.py
 Substitutions:
 [('X', 'Richard')]
 Substitutions:
 [('A', 'y'), ('mother(y)', 'x')]
```

```
exp1 = "knows(A,x)"

exp2 = "knows(y,mother(y))"

substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)
```



Page No: 24

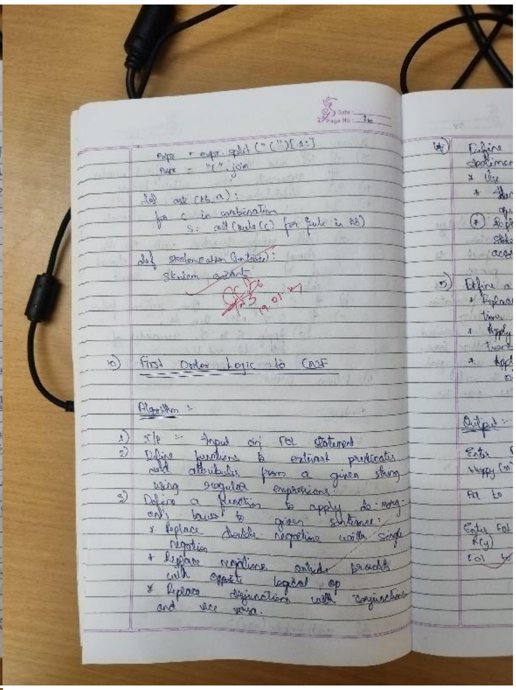
Enter the clause separated by space:
(A∨B) ∧ (C∨D∨J) ∧ (F∨G)
Enter query: A∨B∨C
Trying to prove ((A) ∧ (∨) ∧ (B) ∧ (C) ∧ (∨) ∧ (D)
by contradiction

Knowledge Base entails the query, proved by resolution

9) Implement Unification in First Order Logic

Algorithm:-

```
1) If Ψ1 or Ψ2 is a variable or constant then :
   a) If Ψ1 and Ψ2 are identical then return NIL
   b) Else if Ψ1 is a variable,
      (a) then if Ψ1 occurs in Ψ2, then return failure
      (b) Else return {(Ψ2/Ψ1)}
   c) Else if Ψ2 is a variable,
      (a) If Ψ2 occurs in Ψ1 then return failure
      (b) Else return {(Ψ1/Ψ2)}
   d) Else return failure
2) If the initial predicate symbol in Ψ1 and Ψ2 are not same, then
```



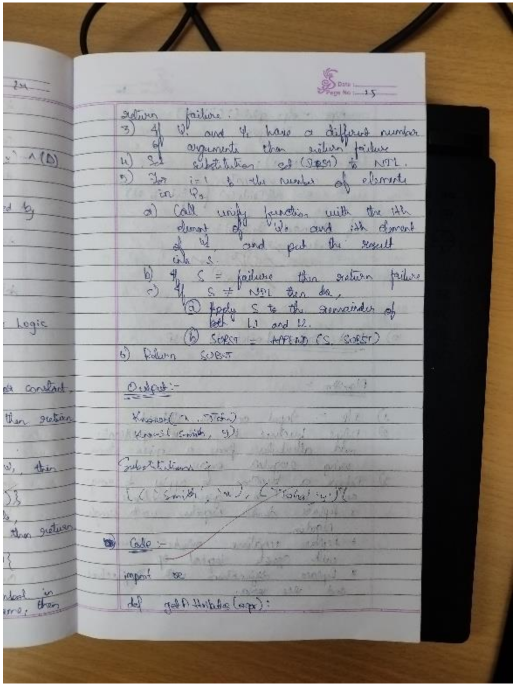
Page No: 26

```
    expr = expr.split("(")[1:]
    expr = "(".join

def ask(KB, a):
    for c in combination
        S= all(rule(c) for rule in KB)

def skolemization(sentence):
    skolem_constant
```

10) First Order Logic to CNF

Algorithm :

```
1) I/P :- Input any FOL statement
2) Define functions to extract predicates and attributes from a given string
   using regular expressions.
3) Define a function to apply de-morgan's laws to given sentence:
   * Replace double negations with single negation
   * Replace negations outside brackets with opposite logical op
   * Replace disjunctions with conjunctions and vice versa.
```



Page No: 25

```
return failure.
3) If Ψ1 and Ψ2 have a different number of arguments then return failure
4) Set substitution set (SUBST) to NIL.
5) For i=1 to the number of elements in Ψ1
   a) Call unify function with the ith element of Ψ1 and ith element
      of Ψ2 and put the result into S.
   b) If S = failure then return failure
   c) If S ≠ NIL then do,
      (a) Apply S to the remainder of both L1 and L2.
      (b) SUBST = APPEND (S, SUBST)
6) Return SUBST
```

Output:-

```
Knows(x, John)
Knows(Smith, y)

Substitutions :
[('Smith', 'x'), ('John', 'y')]
```

Code :-

```
import re

def getAttributes(expr):
```

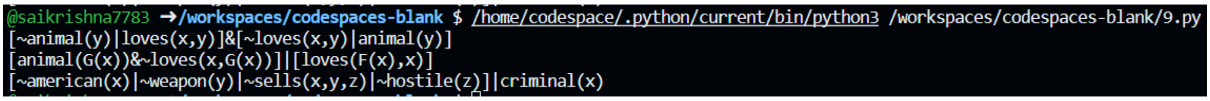
9. Convert a given first order logic statement into Conjunctive Normal Form (CNF).

```
def getAttributes(string):
    expr = '
'
```

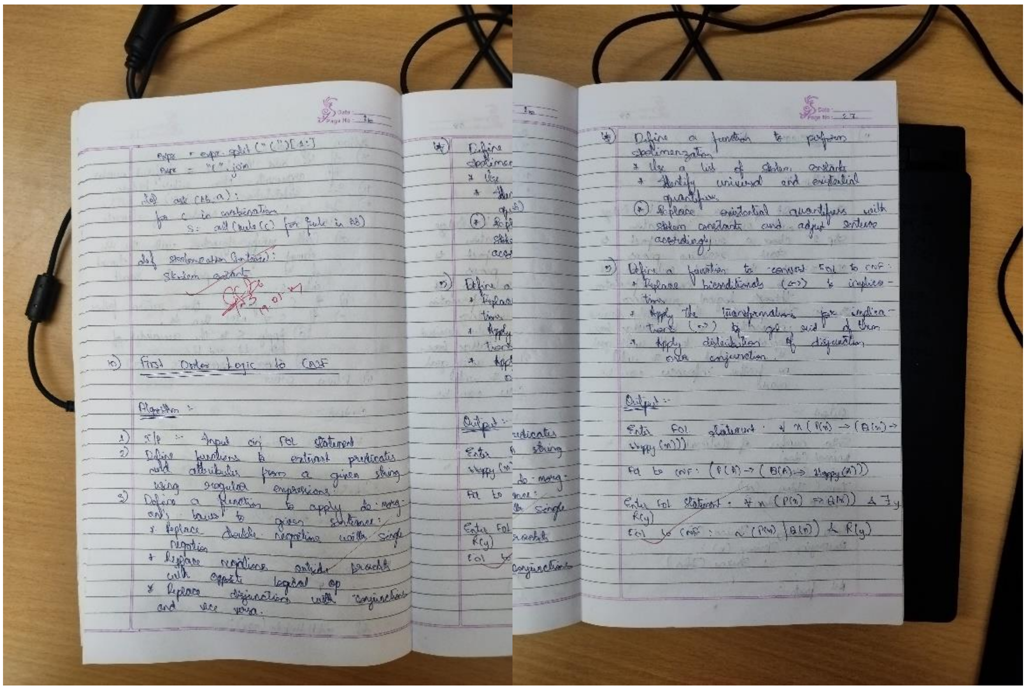
```
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

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

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

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


import re


def fol_to_cnf(fol):


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

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



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

Output:



```
@saikrishna7783 ➜/workspaces/codespaces-blank $ /home/codespace/.python/current/bin/python3 /workspaces/codespaces-blank/9.py
[~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)
```



```
expr = expr.split("(")[1:]
expr = "(".join
def ask(KB, a):
    for c in combination
        s = all(rule(c) for rule in KB)
def skolemization(sentence):
    skolem_constant
```

10) First Order Logic to CNF

Algorithm :-

1) I/p :- Input any FOL statement
2) Define functions to extract predicates and attributes from a given string using regular expressions.
3) Define a function to apply de-morgan's laws to given sentence:
* Replace double negation with single negation
* Replace negations outside brackets with opposite logical op
* Replace disjunctions with conjunctions and vice versa.

4) Define a function to perform skolemization
* Use a list of skolem constants
* Identify universal and existential quantifiers
* Replace existential quantifiers with skolem constants and adjust sentence accordingly
5) Define a function to convert FOL to CNF:
* Replace biconditionals (<->) to implications
* Apply the transformations for implications (->) to get rid of them
* Apply distribution of disjunction over conjunction.

Output :-

Enter FOL statement : ∀ x (P(x) → (Q(x) → happy(x)))

FOL to CNF : (P(A) → (Q(A) → happy(A)))

Enter FOL statement : ∀ x (P(x) => Q(x)) & ∃ y R(y)

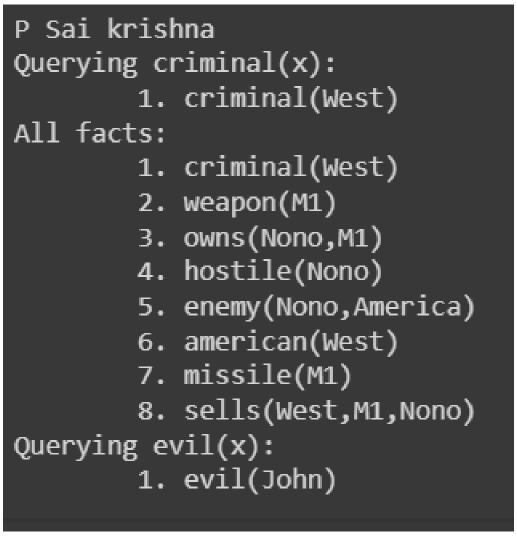
FOL to CNF : ~(P(x) | Q(x)) & R(y)

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

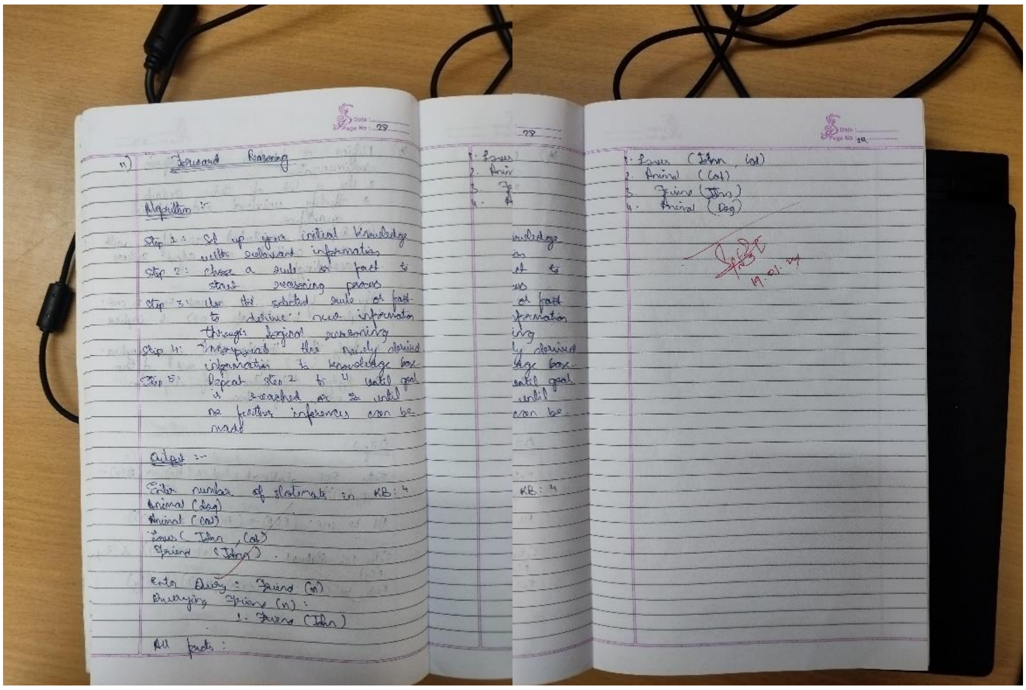
```
import re


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

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

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


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

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

    def getResult(self):
        return self.result

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

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

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

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

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])
        for key in constants:
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate} {attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult() for f in new_lhs]) else None

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

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

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

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

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

Output:



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



Date :
Page No : 28

11) Forward Reasoning

Algorithm :-

Step 1 : Set up your initial knowledge with relevant information
Step 2 : choose a rule or fact to start reasoning process
Step 3 : Use the selected rule or fact to derive new information through logical reasoning
Step 4 : Incorporate the newly derived information to knowledge base.
Step 5 : Repeat step 2 to 4 until goal is reached or until no further inferences can be made

Output :-

```
Enter number of statements in KB: 4
Animal (dog)
Animal (cat)
Loves ( John , Cat)
Friend (John) .

Enter Query : Friend (x)
Querying Friend (x) :
        1. Friend (John)

All facts :
```

Date :
Page No : 29

```
1. Loves (John, Cat)
2. Animal (Cat)
3. Friend (John)
4. Animal (Dog)
```

19.01.24