VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

AI Lab Report

Submitted by

DEEKSHA S (1BM21CS048)

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 to 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 "Internet of things lab" carried out by **DEEKSHA S (1BM21CS048),** 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 2023. The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence lab - (22CS5PCAIN)** work prescribed for the said degree.

Dr. Asha G R
Assistant professor
Department of CSE
BMSCE, Bengaluru

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

Table of Contents

Sl. No.	Title	Page No.
1.	Tic Tac Toe	1-8
2.	8 Puzzle Breadth First Search Algorithm	9-14
3.	8 Puzzle Iterative Deepening Search Algorithm	15-20
4.	8 Puzzle A* Search Algorithm	21-28
5.	Vacuum Cleaner	29-33
6.	Knowledge Base Entailment	34-36
7.	Knowledge Base Resolution	37-39
8.	Simulated Annealing	40-45
9.	Unification	46-52
10.	FOL to CNF	53-57
11.	Forward reasoning	58-63

Program 1 : Tic Tac Toe

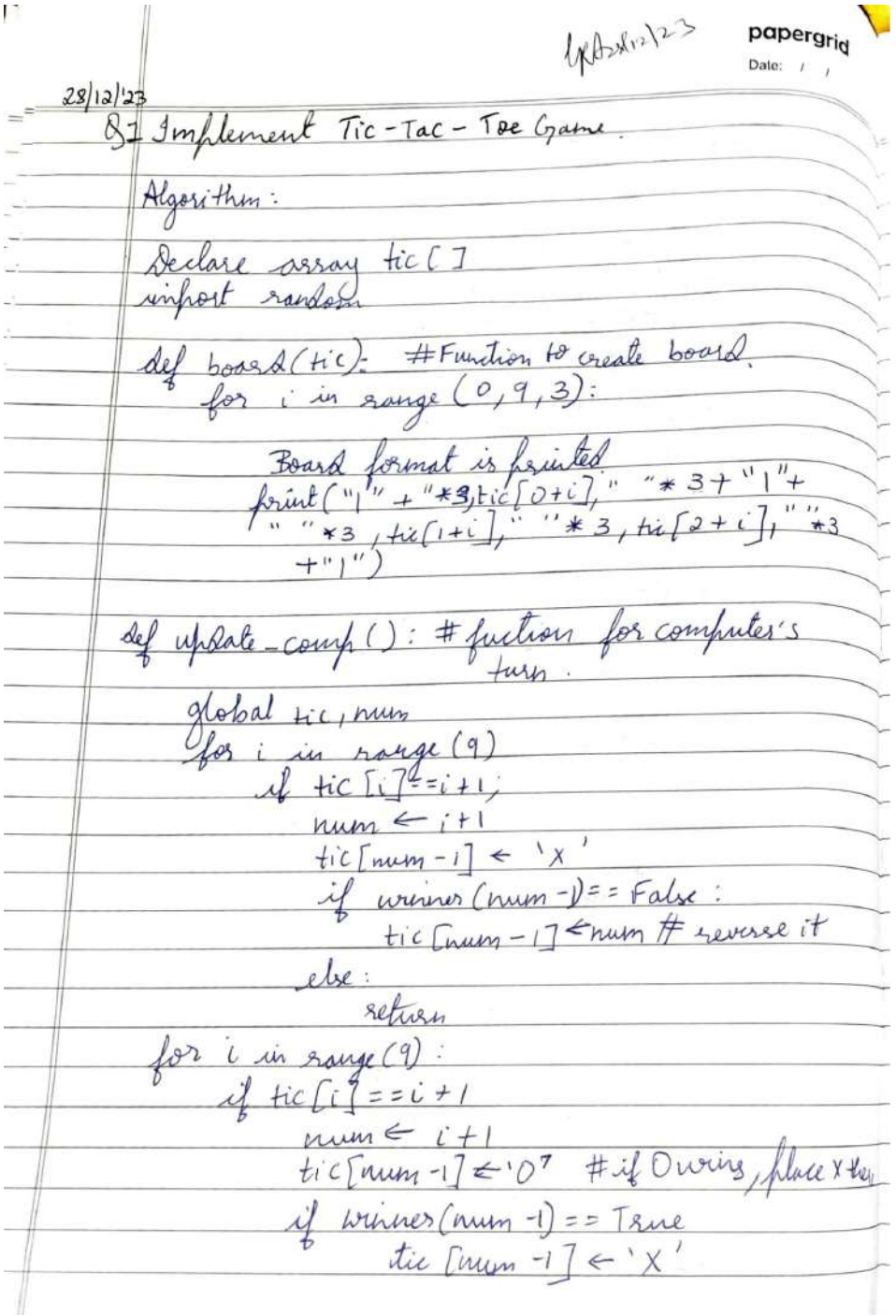
Code:

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


def update_comp():
    global tic,num
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='X'
            if winner(num-1)==False:
                #reverse the change
                tic[num-1]=num
            else:
                return
    for i in range(9):
        if tic[i]==i+1:
            num=i+1
            tic[num-1]='O'
            if winner(num-1)==True:
                tic[num-1]='X'
                return
            else:
                tic[num-1]=num
    num=random.randint(1,9)
    while num  not in tic:
        num=random.randint(1,9)
    else:
        tic[num-1]='X'
```

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

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

try:
    for i in range(1,10):
        tic.append(i)
    count=0
    #print(tic)
    board(tic)
    while count!=9:
        if count%2==0:
            print("computer's turn :")
            update_comp()
            board(tic)
            count+=1
        else:
            print("Your turn :")
            update_user()
            board(tic)
            count+=1
        if count>=5:
```

```
                if winner(num-1):
                    print("winner is ",tic[num-1])
                    break
                else:
                    continue
            except:
                print("\nerror\n")
```

Observation:



28/12/23

Q1 Implement Tic-Tac-Toe Game.

Algorithm:

Declare array tic[ ]
import random

def board(tic): #Function to create board.
    for i in range(0,9,3):
        Board format is printed
        print("|" + "*3,tic[0+i]," "*3+"|"+
        " "*3,tic[1+i]," "*3,tic[2+i]," "*3
        +"|")

def update_comp(): # fuction for computer's turn.
    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 :
                tic[num-1] ←num # reverse it
        else:
            return
    for i in range(9):
        if tic[i]==i+1
            num ← i+1
            tic[num-1] ←'O' # if O wins, place X the
            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 board"))
    while num not in tic:
        num ← int(input("enter a num on board"))
    else
        tic[num-1] ← 'O'

def winner(num)
    if right diagonal or left diagonal is filled
        (0,4) [4,8]                    by same symbol
        return true
    elif any column is filled
        return true
    elif any row is filled
        return true
    else
        return False.

try:
    tic[] = [1,2,3,4,5,6,7,8,9]
    count = 0
```

```
board(tic)
while count != 9:
    if count is even
        # It is computer's turn
        update_comp()
        board(tic)
        count += 1
    else
        Users 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("\n error \n")
```

Working:

Start

↓

Initiate board

↓

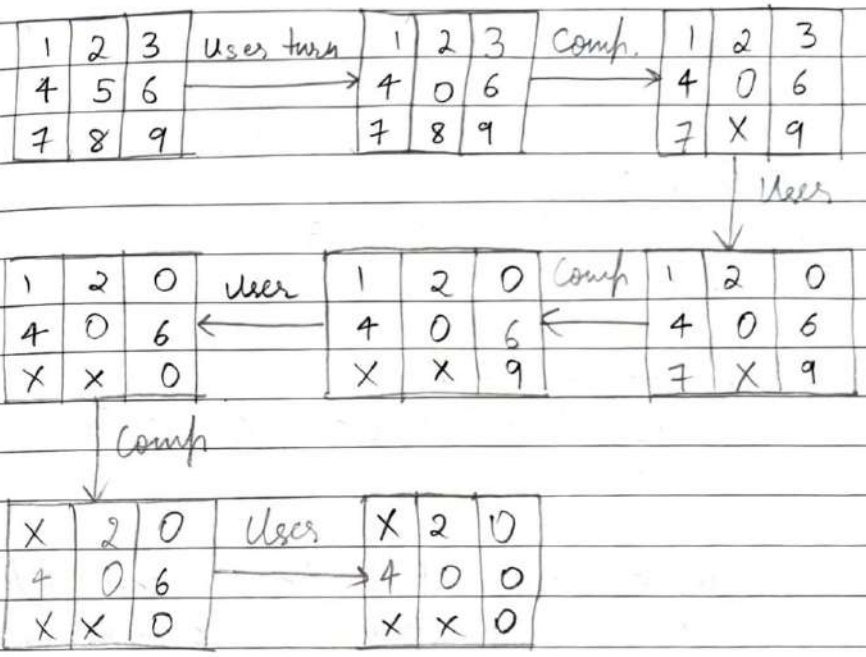
Display board until game ends

↓

→ User's turn

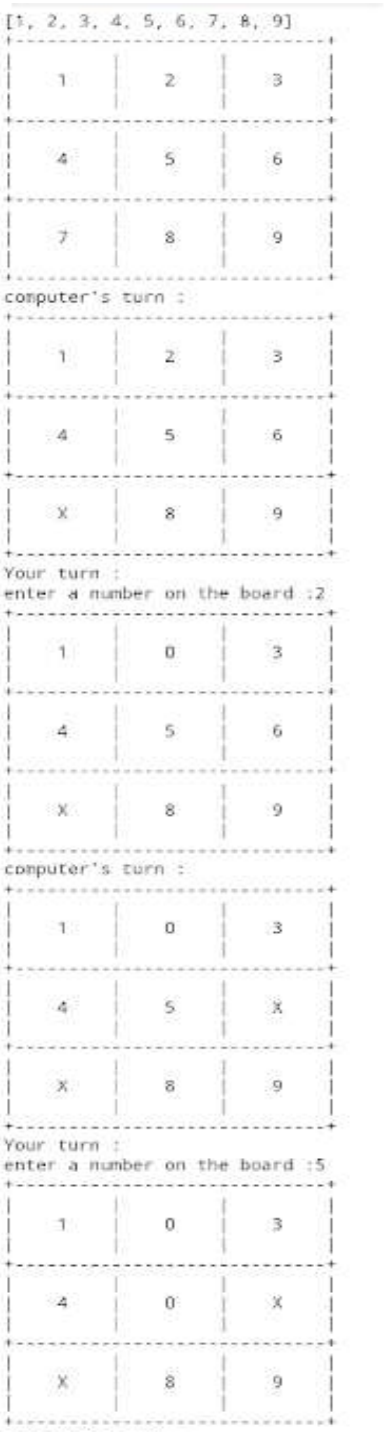
- Place where user asks.
- Validate Winner or not

→ Computer's turn
  • Find winning move and place there
else • Check if opponent wins and place there
else • Place randomly
→ Display board.
End.



O wins!
User won!

Output:


[1, 2, 3, 4, 5, 6, 7, 8, 9]
1 2 3
4 5 6
7 8 9
computer's turn :
1 2 3
4 5 6
X 8 9
Your turn :
enter a number on the board :2
1 0 3
4 5 6
X 8 9
computer's turn :
1 0 3
4 5 X
X 8 9
Your turn :
enter a number on the board :5
1 0 3
4 0 X
X 8 9

```
+---+---+---+
computer's turn :
+---+---+---+
| 1 | 0 | 3 |
+---+---+---+
| 4 | 0 | X |
+---+---+---+
| X | X | 9 |
+---+---+---+
Your turn :
enter a number on the board :9
+---+---+---+
| 1 | 0 | 3 |
+---+---+---+
| 4 | 0 | X |
+---+---+---+
| X | X | 0 |
+---+---+---+
computer's turn :
+---+---+---+
| X | 0 | 3 |
+---+---+---+
| 4 | 0 | X |
+---+---+---+
| X | X | 0 |
+---+---+---+
Your turn :
enter a number on the board :4
+---+---+---+
| X | 0 | 3 |
+---+---+---+
| 0 | 0 | X |
+---+---+---+
| X | X | 0 |
+---+---+---+
computer's turn :
+---+---+---+
| X | 0 | X |
+---+---+---+
|   |   |   |
```

Program 2 : 8 Puzzle Breadth First Search Algorithm

Code:

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

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

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

    #direction array
    d=[]
    if b not in [0,1,2]:
        d.append('u')
    if b not in [6,7,8]:
        d.append('d')
    if b not in [0,3,6]:
        d.append('l')
```

```
    if b not in [2,5,8]:
        d.append('r')

    pos_moves_it_can=[]

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

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

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

Observation:

25/11/23 (A 25-11-23)

Q1. Solve 8puzzle problem using bfs

Algo.

```
bfs(src, target)
    initialize queue[] as array of array to store states to be explored.
    append src to queue
    exp<-[] to store explored states

    while length of queue >0
        Source <- queue.pop(0)
        append source to exp.

        print source

        if sourse == target
            print success

        poss_moves_to_do <-[]
        poss_moves_to_do <-possible_moves(source, exp)

        for move in poss_moves_to_do
            if move is not in exp and move not in queue
                append move into queue.

possible_moves(state, visited_states)

    b <- store index of empty slot
    initialize d[] to store possible directions of b.
```

papergrid

Date: / /

if b not in [0,1,2] (first row)
append 'u' in d[] → means we can move empty slot up.

if b not in [6,7,8] (last row)
append 'd' in d[]

if b not in first column
append 'l' in d[]

if b not in last column
append 'r' in d[]

initialize pos_moves_it_can = []
for i in d:
append (gen(state, i, b)) to pos_moves_it_can

return the moves in pos_moves_it_can which is not in visited_states.

gen (state, m, b)
temp ← copy state.
if (m == 'd') if move is down
swap temp[b+3] & temp[b]
if move is up
swap temp[b-3] & temp[b]
if move is left
swap temp[b-1] and temp[b]
if move is right
swap temp[b+1] & temp[b]
return temp

Example.

src = [1, 2, 3, 4, 5, 6, 0, 7, 8]

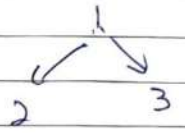
representation

1	2	3
4	5	6
0	7	8

0 represents empty slot.

1	2	3	
4	5	6	target
7	8	0	

target = [1, 2, 3, 4, 5, 6, 7, 8, 0]



→ bfs

( )

1 2 3 ~~4 5 6 0~~

4 5 6 b = 6

6 0 7 8 d =

bfs(src, target)

1	2	3
4	5	6
0	7	8

1	2	3
0	5	6
4	7	8

up

1	2	3
4	5	6
7	0	8

right

Tree will keep getting explored.
We stop when
State == target.

Output:

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

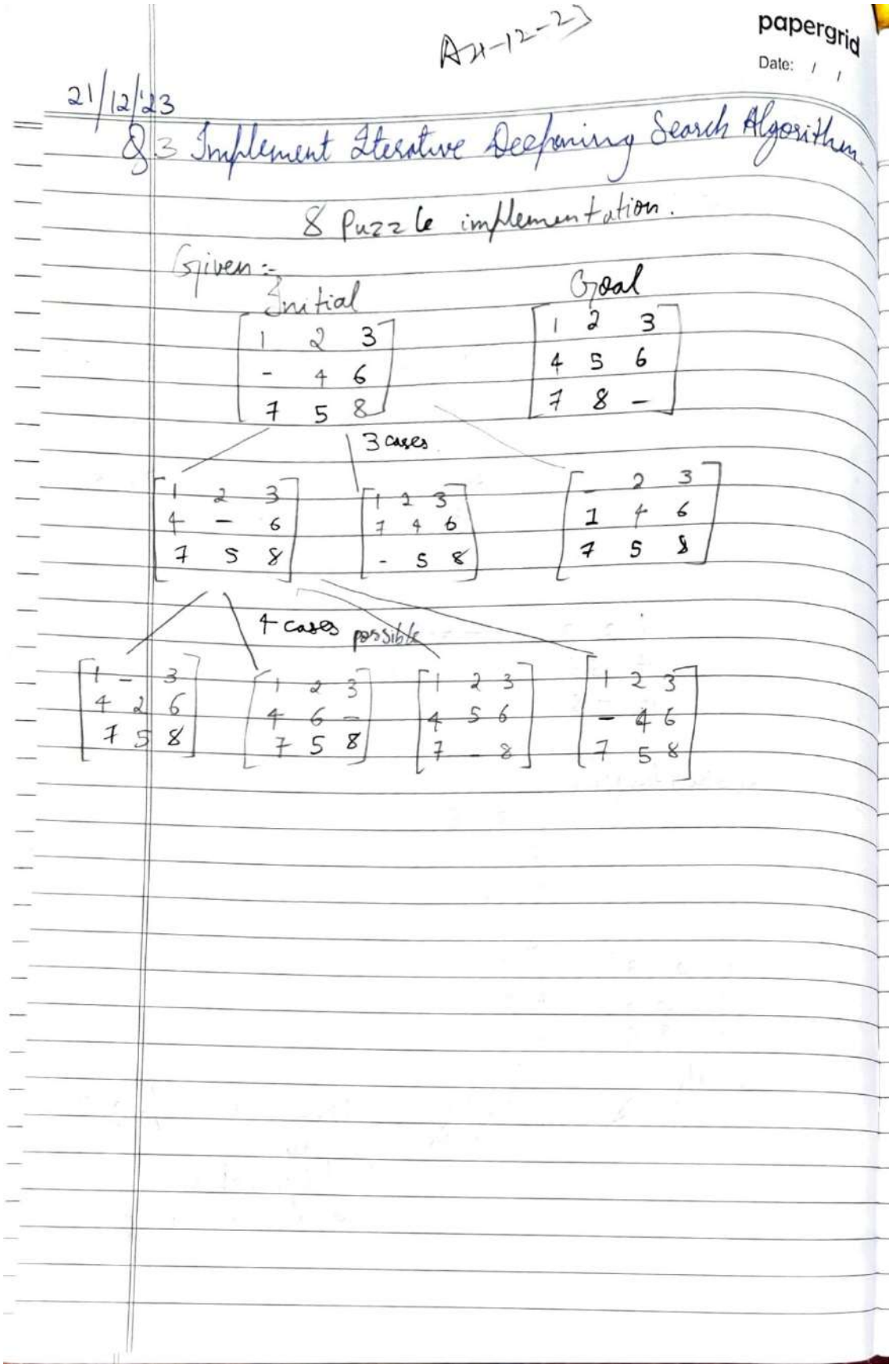
Program 3 : 8 Puzzle Iterative Deepening Search Algorithm

Code:

```
# 8 Puzzle problem using Iterative deepening depth first search algorithm

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

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



def possible_moves(state):
    b = state.index(0)  # ) indicates White space -> so b has index of it.
    d = []  # direction

    if b not in [0, 1, 2]:
        d.append('u')
    if b not in [6, 7, 8]:
        d.append('d')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
```

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


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

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

    return temp


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

route = id_dfs(initial, goal, possible_moves)

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

Observation:



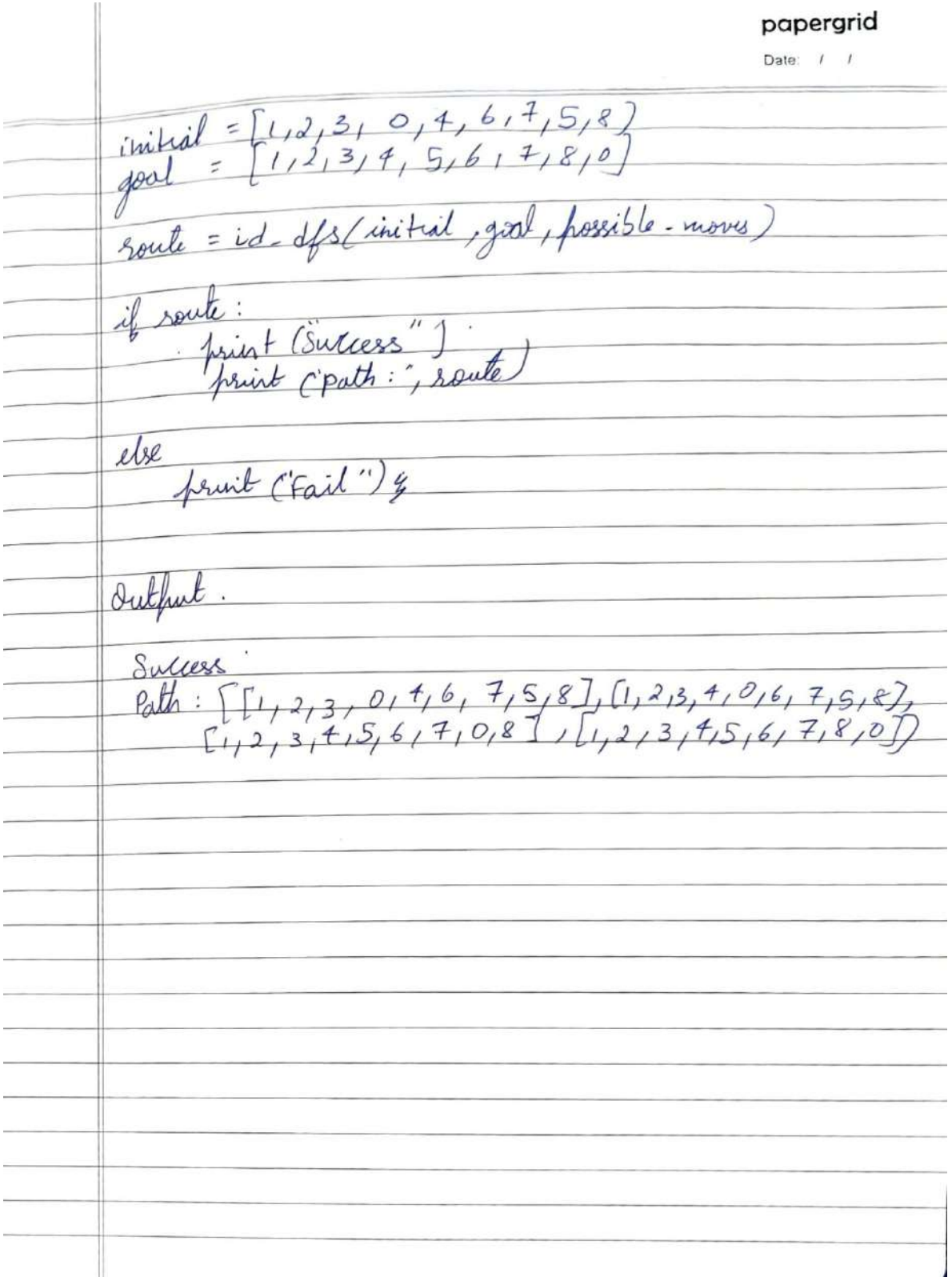
papergrid

Date: / /

8 puzzle problem. Algorithm using iterative deepening.

```
def id_dfs(puzzle, goal, get_moves):
    import itertools

    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 routes:
                next_route ← dfs(route + [move],
                if next_route:  depth - 1)
                then return next_route

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

~~def generate(state, m, b):~~
~~    temp ← State.copy()~~

~~    if m == 'd'~~
def possible_moves(state):
    b ← state.index(0)  Stores index of empty slot
    initialize d[] to store possible directions of b.

    if b ~~not~~ not in [0, 1, 2]: (first row)
```

```
append 'u' in d[]

if b not in [6,7,8]:
    append 'd' in d[]

if b not in [0,3,6]
    append 'l' in d[]

if b not in last column
    append 'r' in d[]

pos_moves[] declared

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] swapped

    if m == 'u'
        swap temp[b-3] & temp[b]
if move is left
    swap temp[b-1] & temp[b]
if move is right
    swap temp[b+1] & temp[b]

return temp
```



Output:

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

Program 4 : 8 Puzzle A* Search Algorithm

Code:

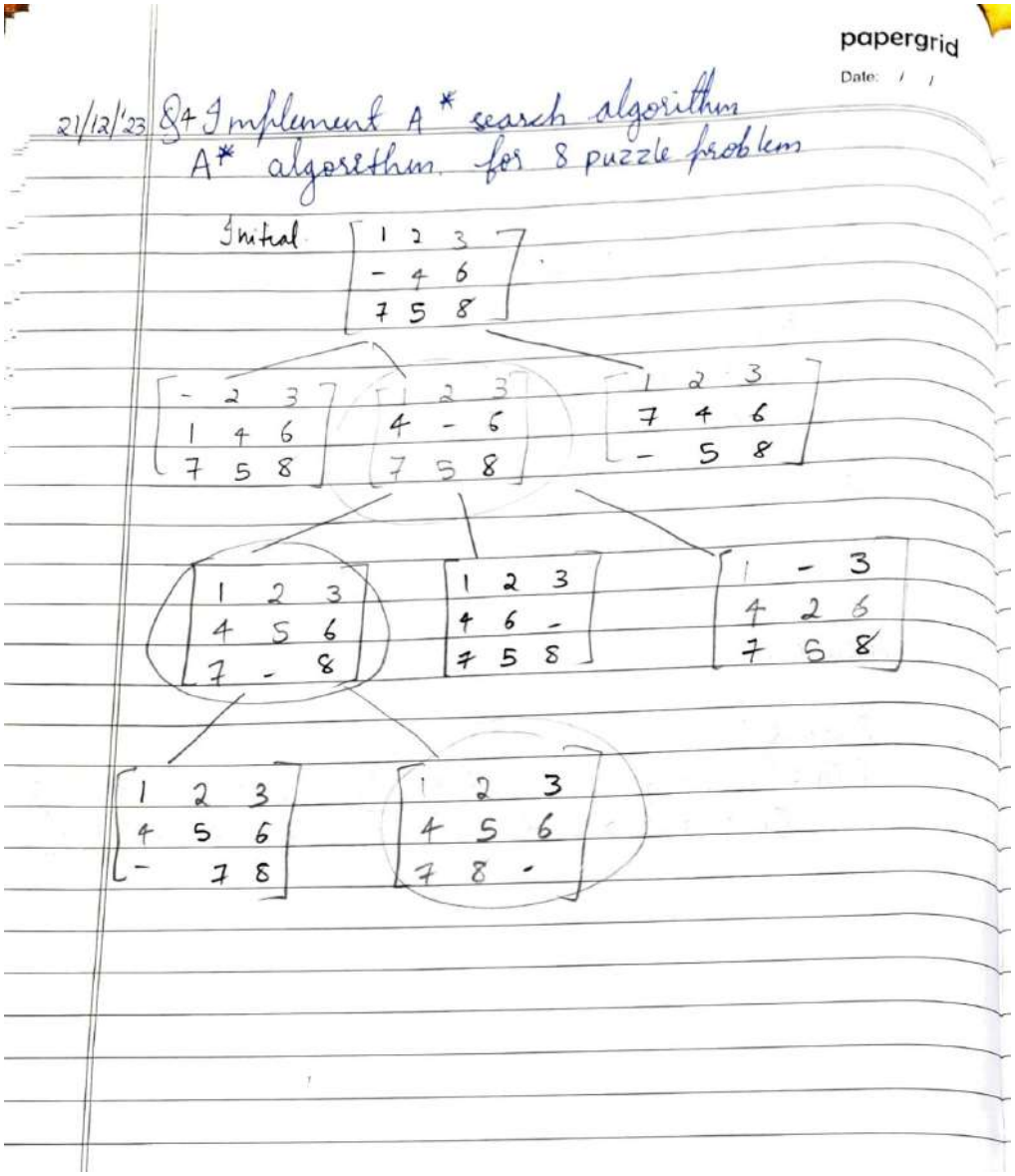
```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

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

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

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

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


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

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

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

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


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

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

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

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


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

Observation:



Algorithm:

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

Above is the initialization.

def generate_child(self): to generate state space tree

x, y ← self.find(self.data, '_') to find black space.

val_list ← [[x, y-1], [x, y+1], [x-1, y], [x+1, y]] has 4 directions

children ← [] declared.

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):
to move the puzzle pieces
if x2 >= 0 and x2 < len(self.data) and
y2 >= 0 and y2 < len(self.data):
temp_puz [] array declared.

```
        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


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 rang(0, self.n)
            if start[i][j] = goal[i][j]
                and start[i][j] != '_':
                    temp += 1
    return temp.
```

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

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

        for i in cur.data:
            for j in i:
                print(j, end = " ")

        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()
```

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 _
```

Program 5 : Vacuum Cleaner

Code:

```
def clean_room(floor, room_row, room_col):
    if floor[room_row][room_col] == 1:
        print(f"Cleaning Room at ({room_row + 1}, {room_col + 1}) (Room was dirty)")
        floor[room_row][room_col] = 0
        print("Room is now clean.")
    else:
        print(f"Room at ({room_row + 1}, {room_col + 1}) is already clean.")

def main():
    rows = 2
    cols = 2
    floor = [[0, 0], [0, 0]]  # Initialize a 2x2 floor with clean rooms

    for i in range(rows):
        for j in range(cols):
            status = int(input(f"Enter clean status for Room at ({i + 1}, {j + 1}) (1 for dirty, 0 for
clean): "))
            floor[i][j] = status

    for i in range(rows):
        for j in range(cols):
            clean_room(floor, i, j)

    print("Returning to Room at (1, 1) to check if it has become dirty again:")
    clean_room(floor, 0, 0)  # Checking Room at (1, 1) after cleaning all rooms

if __name__ == "__main__":
    main()
```

Four rooms:

```
def clean_room(room_name, is_dirty):
   if is_dirty:
      print(f"Cleaning {room_name} (Room was dirty)")
      print(f"{room_name} is now clean.")
```

```
        return 0  # Updated status after cleaning
    else:
        print(f"{room_name} is already clean.")
        return 0  # Status remains clean

def main():
    rooms = ["Room 1", "Room 2"]
    room_statuses = []

    for room in rooms:
        status = int(input(f"Enter clean status for {room} (1 for dirty, 0 for clean): "))
        room_statuses.append((room, status))
    print(room_statuses)

    for i, (room, status) in enumerate(room_statuses):
        room_statuses[i] = (room,clean_room(room, status)) # Update status after cleaning

    print(f"Returning to {rooms[0]} to check if it has become dirty again:")
    room_statuses[0]=status = (rooms[0],clean_room(rooms[0], room_statuses[0][1])) # Checking
Room 1 after cleaning all rooms

    print(f"{rooms[0]} is {'dirty' if room_statuses[0][1] else 'clean'} after checking.")

if __name__ == "__main__":
    main()
```

Observation:

papergrid

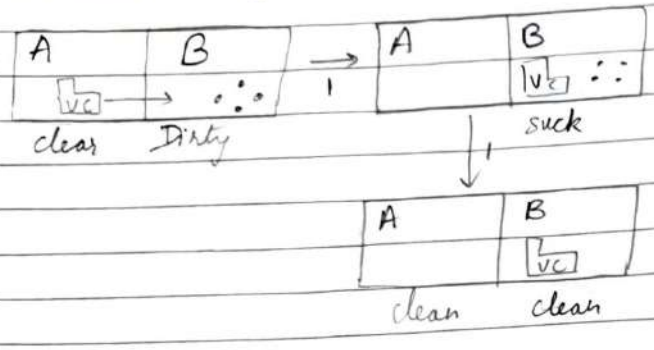
Date: / /

28/12/'23

6. Implement vacuum cleaner agent.

2 locations A & B

Example:



Total cost = 2.

Always goal = {'A': '0', 'B': '0'}

Algorithm:

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

    loc ⇐ input("Enter loc of vaccum")
    status ⇐ Take status of loc from user
    status 2 ⇐ Take status of other loc

if loc == 'A'
        vacuum in location A
        if status == '1'
            A is Dirty
            cost += 1 #to suck
            loc A is cleaned

if status 2 == '1'
        loc B is dirty.
```

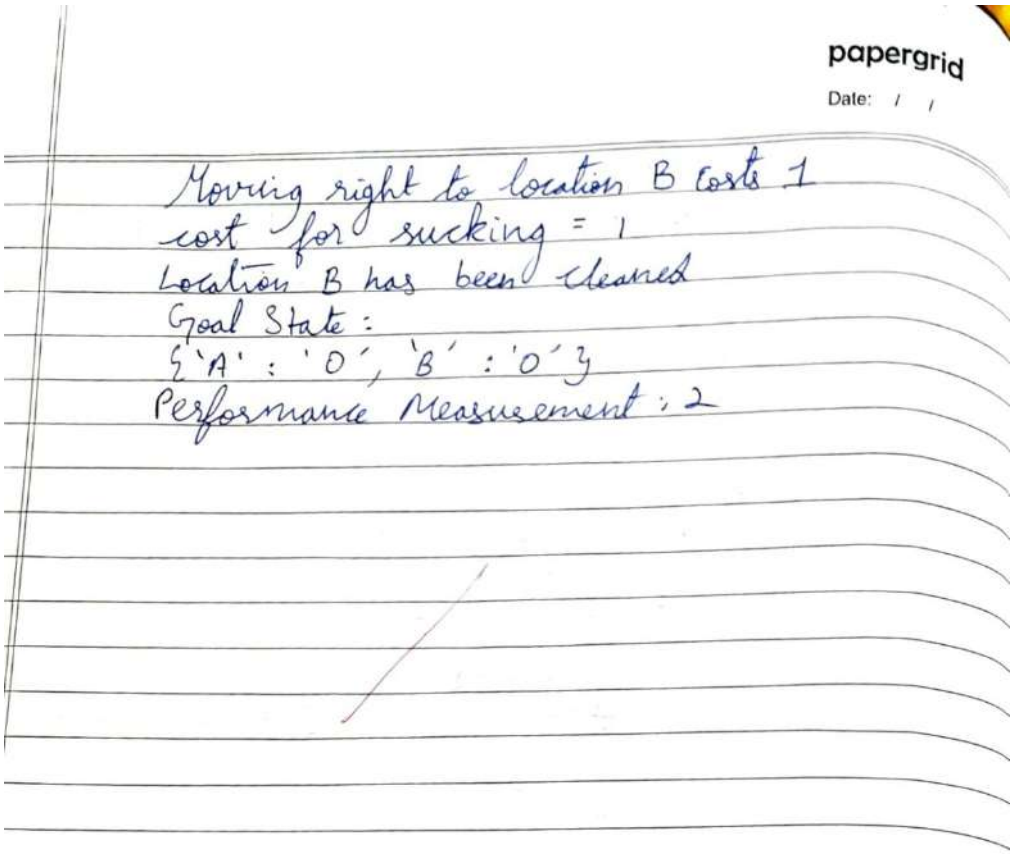
```
        cost += 2
        # To move right and clean
        location B is cleaned

    else
        no action

if status == '0'
    A is already clean
    if status2 == '1'
        B is dirty
        cost += 2
        # To move to B and clean
    else
        no action
```

Repeat the same steps when intial position of vaccum cleaner is B.

```
print ("Goal State :")
print (goal_state)
print ("performance measure :" + str(cost))

vacuumworld()
```

Output:

```
Enter location of vaccum: A
Enter status of A : 0
Enter status of other room : 1
Vaccum is placed in Location A
Location A is already clean
Location B is dirty
```



Output:

```
0 indicates clean and 1 indicates dirty
Enter Location of VacuumA
Enter status of A1
Enter status of other room0
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
No action1
Location B is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 1
```

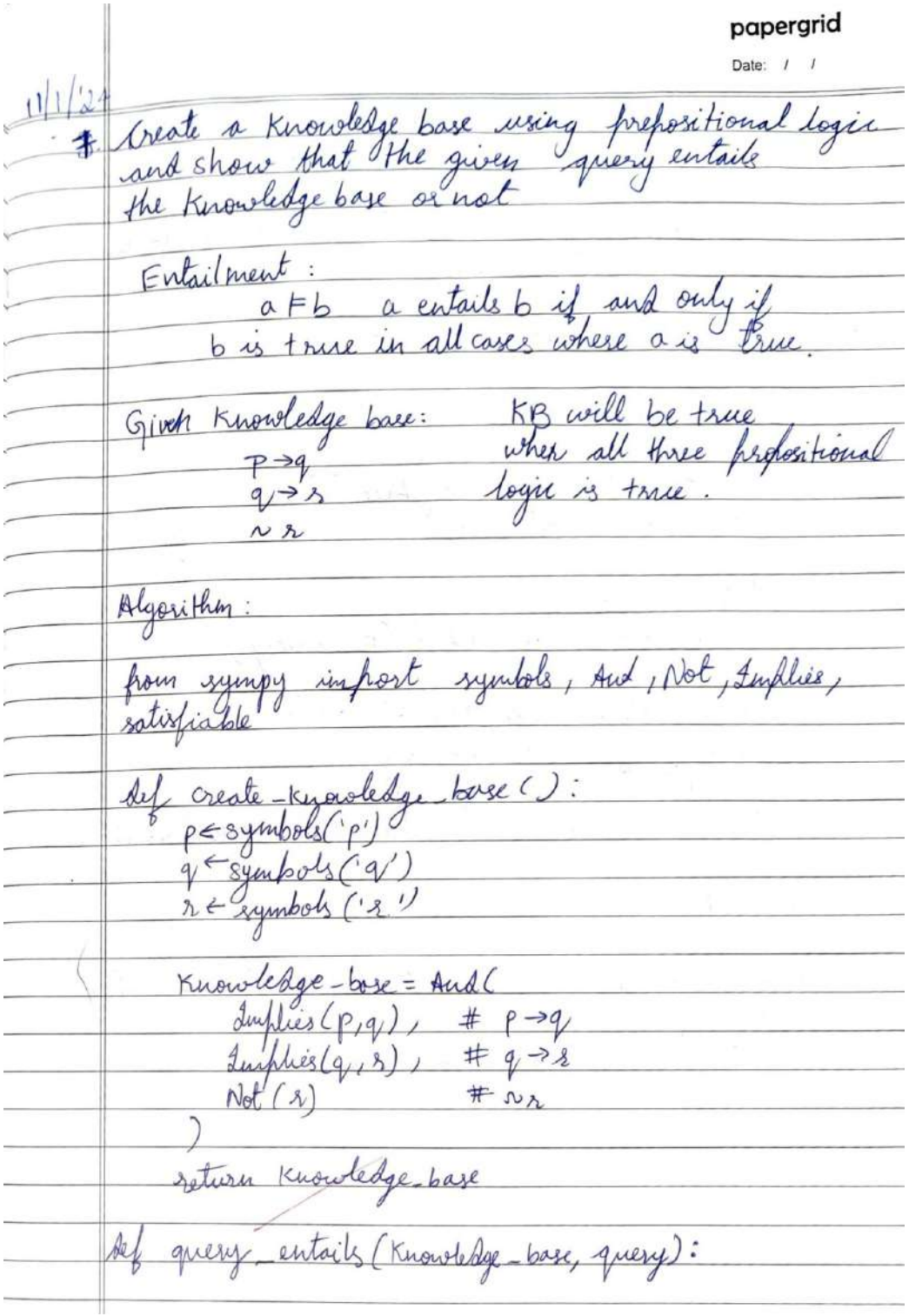
Program 6 : Knowledge Base Entailment

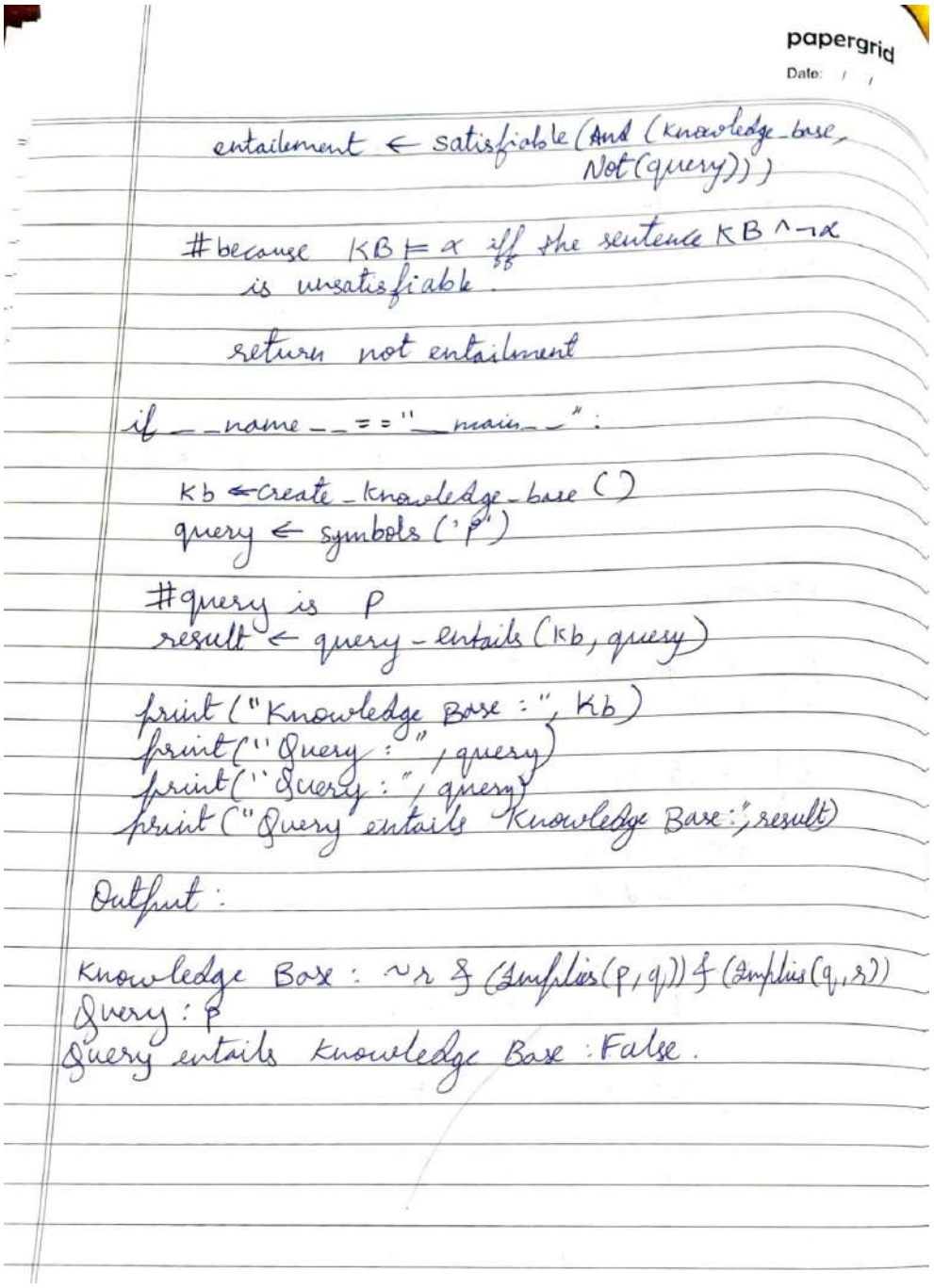
Code:

```
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)
```

Observation:





Output:

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

Program 7 : Knowledge Base Resolution

Code:

```
def tell(kb, rule):
   kb.append(rule)

combinations = [(True, True, True), (True, True, False),
                (True, False, True), (True, False, False),
                (False, True, True), (False, True, False),
                (False, False, True), (False, False, False)]



def ask(kb, q):
   for c in combinations:
      s = all(rule(c) for rule in kb)
      f = q(c)
      print(s, f)
      if s != f and s != False:
         return 'Does not entail'
   return 'Entails'



kb = []

# Get user input for Rule 1
rule_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and
x[1]): ")
r1 = eval(rule_str)
tell(kb, r1)

# Get user input for Query
query_str = input("Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or
x[1]): ")
q = eval(query_str)

# Ask KB Query
result = ask(kb, q)
print(result)
```

Observation:

papergrid

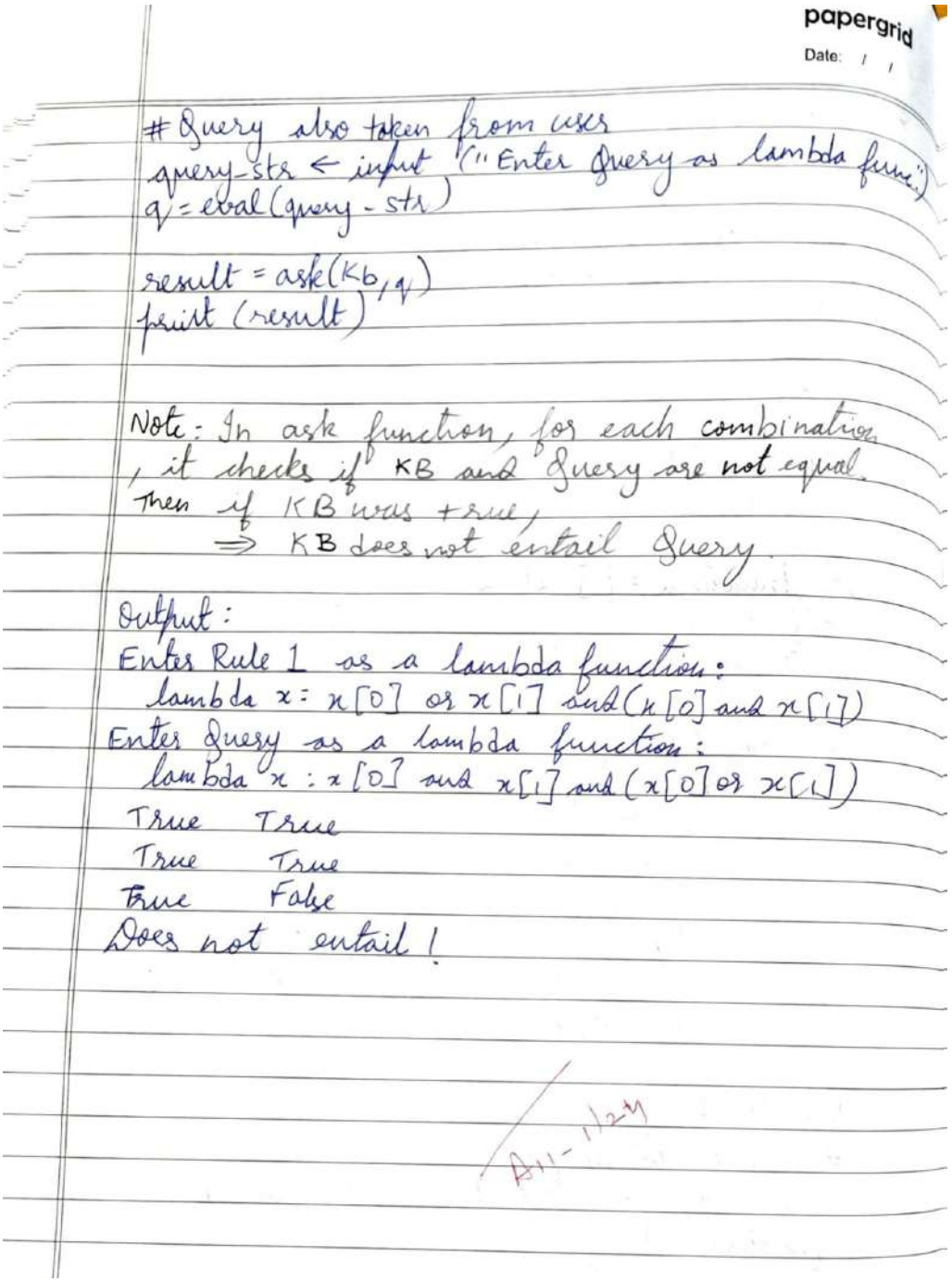
Date: / /

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

Resolution combines two clauses containing complementary literals and generates a new clause.

Code:

```
def tell(kb, rule):
    kb.append(rule)

combinations = [] It is an array containing all possible values for 3 variables (True or False)

def ask(kb, q):
    for c in combinations:
        s ← all(rule(c) for rule in kb)
        f ← q(c)
        print(s, f)
        if s != f and s != False:
            return 'Does not entail'
    return 'Entails'

kb ← [ ]
#input taken from user
rule_str ← input("Enter Rule 1 as a lam function")
r1 ← eval(rule_str)
tell(kb, r1)
```



Output:

```
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] and (x[0] and x[1])
Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1])): lambda x: x[0] and x[1] or x[2]
True True
True True
True True
True False
Does not entail
```

Program 8. Simulated Annealing

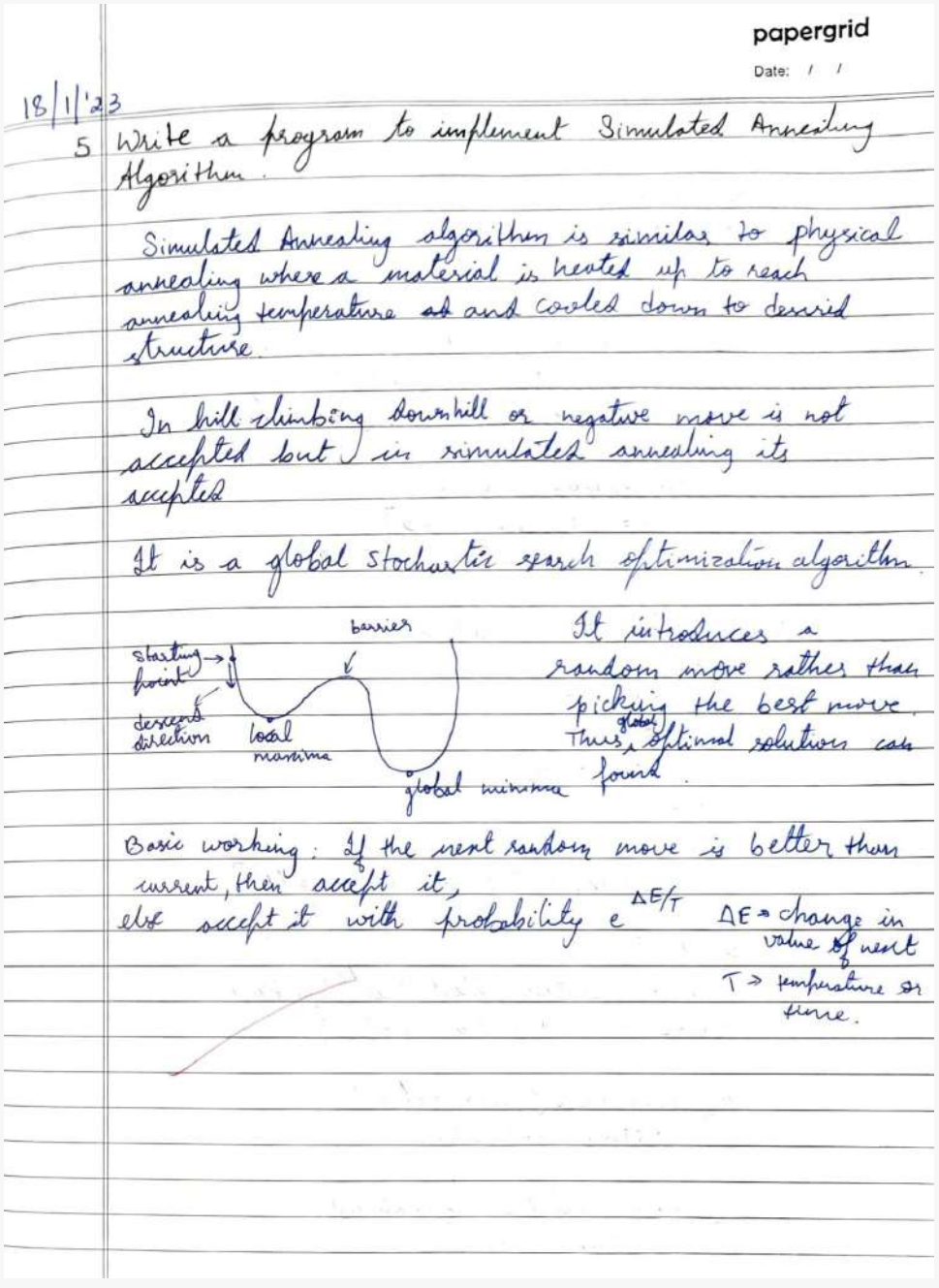
Code:

```
import random
import math

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

# Function prototype
def gen_rand_sol():
    a = [1, 2, 3, 4, 5]
    return Solution(-1.0, a)

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

def neighbor(current_sol):
    return current_sol
```

```
def cost(input_configuration):
    return -1.0

# Mapping from [0, M*N] --> [0, M]x[0, N]
def index_to_points(index):
    return [index % M, index // M]

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

print(mini.CVRMSE, "\n")

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

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

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

Observation:



papergrid

Date: / /

18/1/'23

5 Write a program to implement Simulated Annealing Algorithm.

Simulated Annealing algorithm is similar to physical annealing where a material is heated up to reach annealing temperature ~~at~~ and cooled down to desired structure.

In hill climbing downhill or negative move is not accepted but in simulated annealing its accepted

It is a global stochastic search optimization algorithm.

It introduces a random move rather than picking the best move. Thus, global optimal solutions can found.

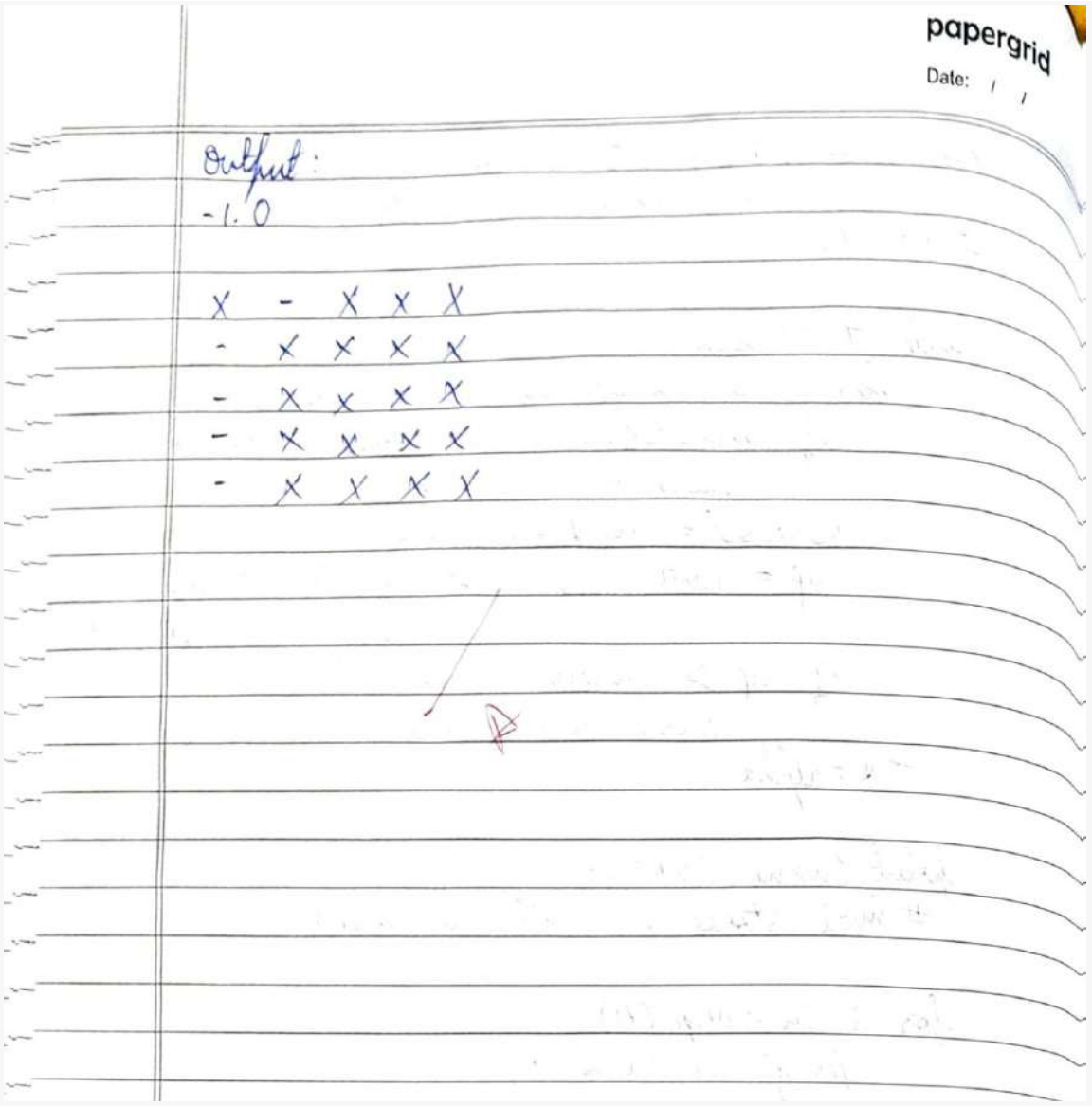
Basic working: If the next random move is better than current, then accept it, else accept it with probability $e^{\Delta E/T}$ ΔE = change in value of next

T → temperature or time.

Algorithm

```
import random
import math

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

    def gen_rand_sol():
        a = [1,2,3,4,5]
        return Solution(-1.0, a)

# Initialize variables
T = 1
Tmin = 0.0001
alpha = 0.9
no_iterations = 100
M = 5
N = 5
source_array = [['x' for _ in range(N)]
                for _ in range(M)]
temp = []
mini = Solution(float('inf'), temp)
curr_sol = gen_ran_sol()

def neighbour(curr_sol):
    return curr_sol

def cost(input_configuration):
    return -1.0
```

```
def index_to_points(index)
    returns [index % M, index // M]
#1D to 2D

while T > Tmin:
    for _ in range(no_of_iterations):
        if curr_sol.CVRMSE < mini.CVRMSE:
            mini ← curr_sol
        new_sol ← neighbour(curr_sol)
        ap ← math.exp((curr_sol.CVRMSE-
                       new_sol.CVRMSE)/T)
        if ap > random.random():
            curr_sol = new_sol
    T *= alpha

print(mini.CVRMSE, "\n")
#mini stores the optimum solution.

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

for obj in mini.config:
    coord ← index_to_points(obj)
    source_array[coord[0]][coord[1]] = '-'

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



Output:

```
-1.0

X - X X X
- X X X X
- X X X X
- X X X X
- X X X X
```

Program 9 : Unification

Code:

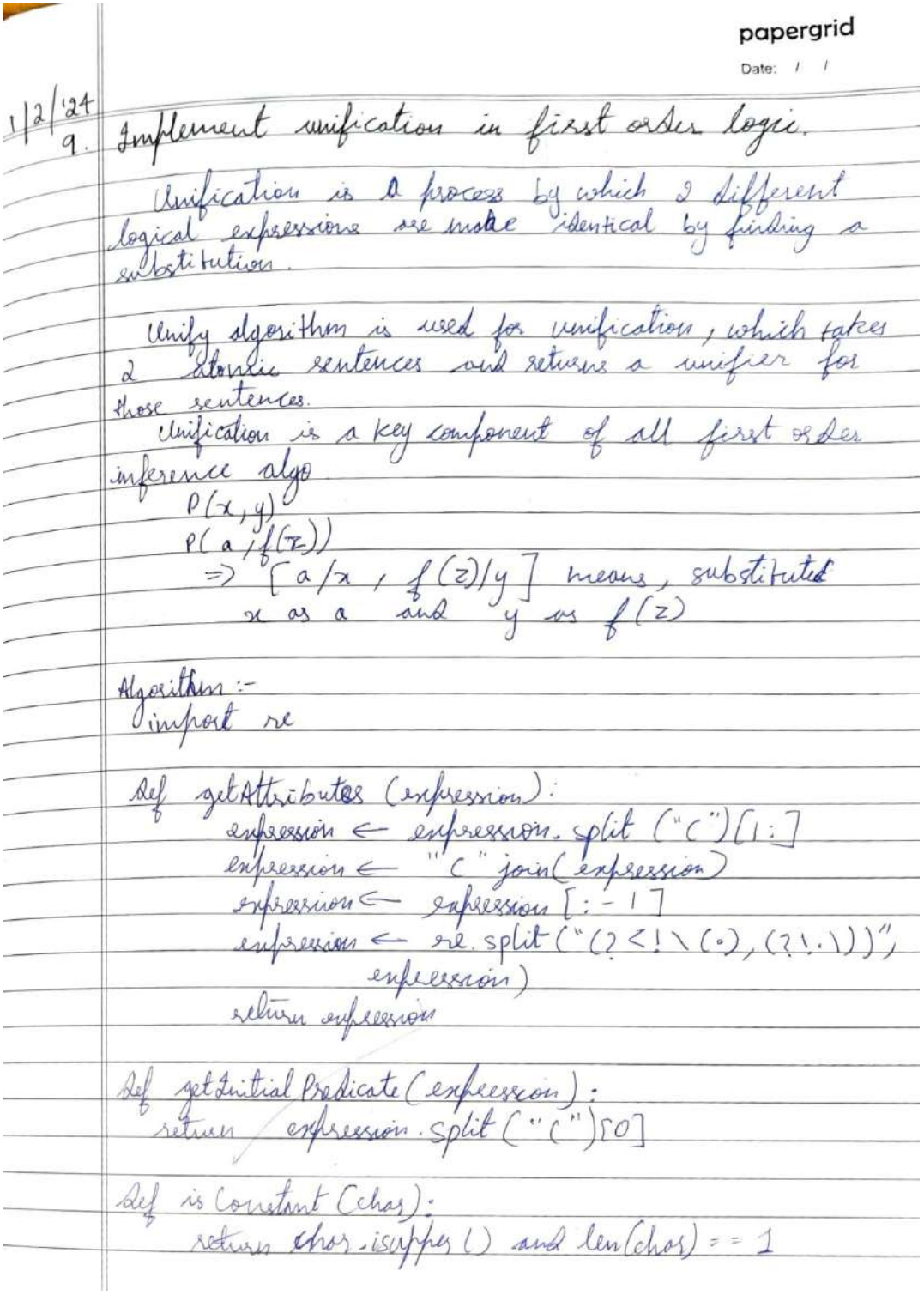
```
import re

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)
exp1 = "knows(A,x)"
exp2 = "knows(y,mother(y))"
```

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

Observation :



1/2/24
9. Implement unification in first order logic.

Unification is a process by which 2 different logical expressions are make identical by finding a substitution.

Unify algorithm is used for unification, which takes 2 atomic sentences and returns a unifier for those sentences.
Unification is a key component of all first order inference algo

P(x, y)
P(a, f(z))
=> [a/x, f(z)/y] means, substituted x as a and y as f(z)

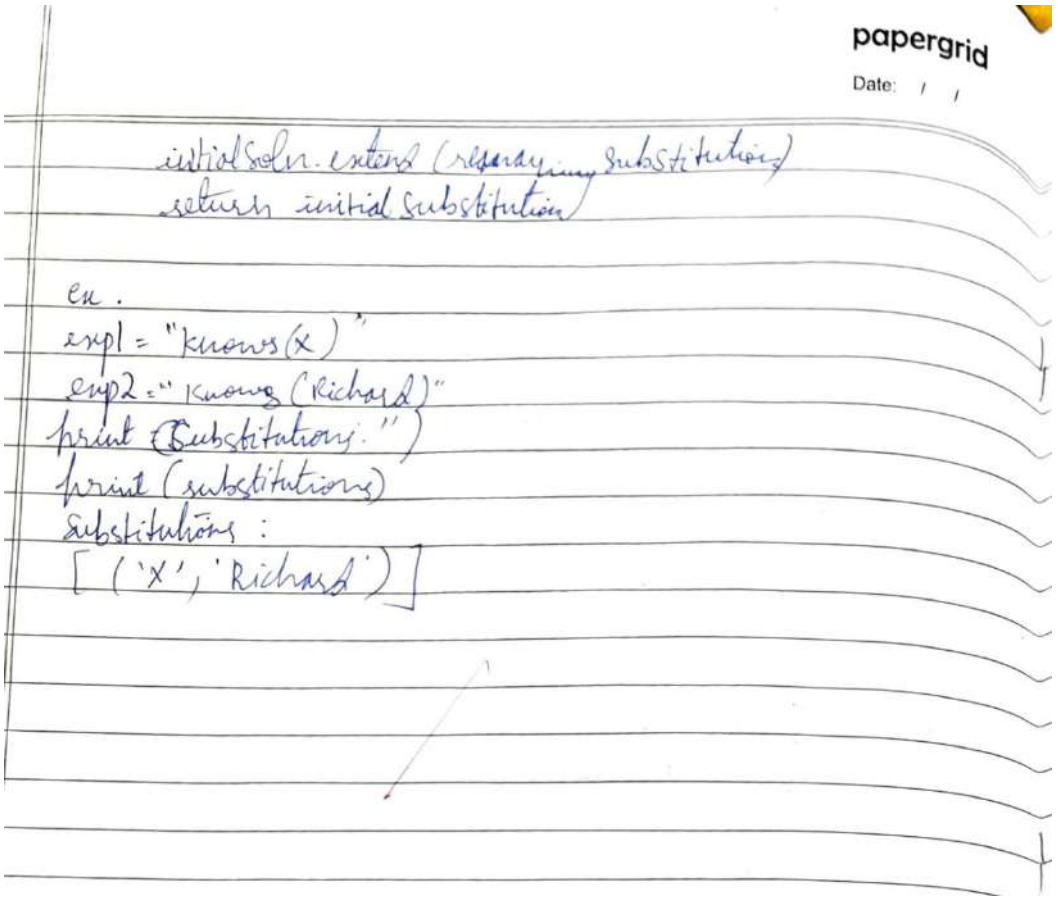
Algorithm :-

```
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
```

Date: / /

```
Def isVariable(char):
    returns char.islower() and length is 1

Def replaceAttributes(exp, old, new):
    attributes ← getAttributes(exp)
    for index val in enumerate(attributes):
        if val equal to old
            attributes[index] ← new
    predicate ← getInitialPredicate(exp)
    return predicate + "(" + "," join(attributes) + ")"

Def apply(exp, substitutions):
    for substitution in substitutions:
        new, old ← situation
        exp ← replaceAttributes(exp, old, new)
    return exp

Def checkOccurs(var, exp):
    check if var is present in x

Def getFirstPart(expression)
    returns the first attribute in the expression

Def getRemainingPart(expression)
    get attribute from initial predicate
    return predicate + "(" + ",".join attributes[1:] + ")"

Def unify(exp1, exp2)
    if both are same returns []
    if both are constant return false
    if exp1 is constant
        return [(exp1, exp2)]
```

```
if exp2 is constant
    return [(exp2, exp1)]
if exp1 is variable
    if exp1 occurs in exp2
        return False
    else:
        return [(exp2, exp1)]
if exp2 is variable
    if exp2 occurs in exp1
        return False
    else
        return [(exp1, exp2)]
if initialPredicates don't match
    return false
if length of attributes of exp1 & exp2 don't match
    return false

initialSoln ← unify(first part of exp1 and exp2)
if not initialSoln
    return false
if attributeCount1 == 1)
    return initial soln
tail1 ← getRemainingPart(exp1)
tail2 ← getRemainingPart(exp2)
if initialsoln1 = [ ]
    tail1 ← apply(tail1, initialSoln)
    tail2 ← apply(tail2, initialSoln)

remaining ← unify(tail1, tail2)
    if not
        return false
```



Output:

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

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

Program 10 : FOL to CNF

Code:

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

def getPredicates(string):
    expr = '[a-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)"))
```

Observation:

papergrid

Date: / /

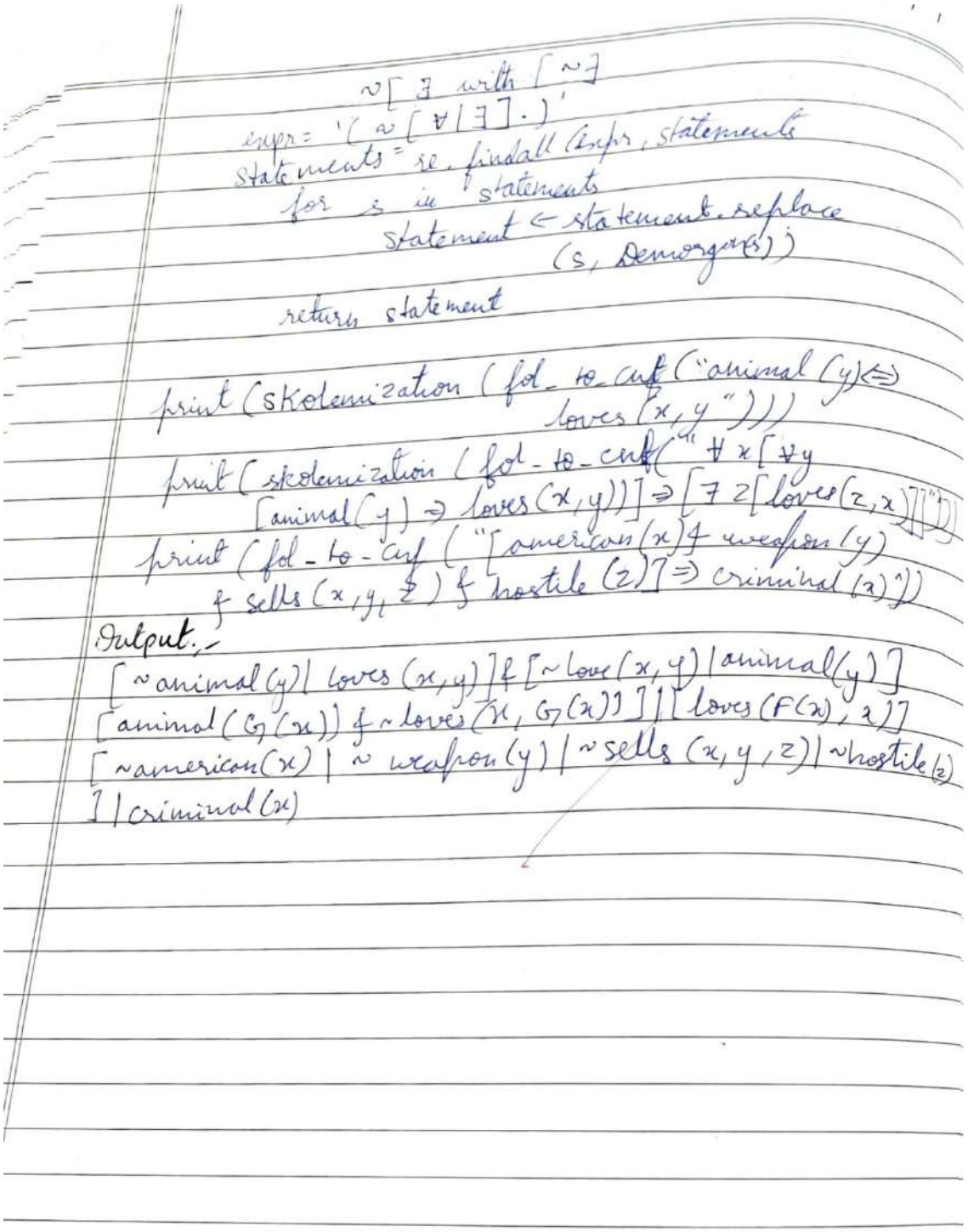
10. Convert a given first order logic statement into Conjunctive Normal Form (CNF)

Conjunctive normal form CNF is an approach to express FOL with AND of OR

ex $A_1 \wedge A_2 \wedge A_3 \ldots \wedge A_n$

ex $(A \vee B) \wedge (C \vee D)$

```
def DeMorgon (sentence)
    replace (predicate, '~ {predicate}')
    also replace OR with AND
                 AND with OR
    join string list
    replace ('~~a' with 'a')
    return if '[{String}]' if flag else String

def SKOLEMIZATION (sentence):
    create a list of SICOLEM CONSTANTS

def fol_to_cnf (fol):
    statement ← fol.replace ("<=>", "-")
    replace with ⇒ in the statement
    statement ← statement.replace ("⇒", "_")
    if '[' is in S and ']' not in S
                add ']'
    for ~~sin~~ S in statements:
            statement ← statement.replace (S, fol_to_cnf())

convert A → B into ~A∨B
replace ~∨ with ] statements (post) ~
replace ~] with ∨ statement (post) ~
replace ~[∀ with ∃[~∀
```



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)
```

Program 11 : 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 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()

kb_ = KB()
kb_.tell('king(x)&greedy(x)=>evil(x)')
kb_.tell('king(John)')
kb_.tell('greedy(John)')
kb_.tell('king(Richard)')
kb_.query('evil(x)')
```

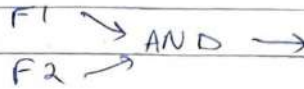
Observation:

papergria

Date: / /

11. Create a KB consisting of FOL statements and prove the given query using forward reasoning

Forward reasoning is a top-down approach that starts with available facts and uses logical rules and heuristics to derive conclusions and make decisions




```
def isvariable
    return len(x) == 1 and x is lower and x is alpha

def getAttribute(strings)
    return attributes in the string

def getPredicates(string)
    return all predicates in string

class Fact:
    initialize expression, predicate, params and
    result
    def splitExpression(self, expression)
        return predicates and params of the expression

def getResults(self)
    return self.result

def getConstants(self)
    returns constants in params

def getvariables(self)
    return variables in params
```

```
Def substitute (self, constants)
    return the facts

class Implication:
    initialize variables
    in lhs if val.predicate == fact.predicate
        constants[v] <- fact.getConstants
    predicate.attributes = getPredicates()
                           str(getattributes())
    return Fact(expr) if (new-lhs) and
        f.getResults
```

Create a KB

```
ex: kb_ <= KB()
    kb_.tell('King(x) & greedy(x) => evil(x)')
    kb_.tell('King(John)')
    kb_.tell('greedy(John)')
    kb_.tell('King(Richard)')
    kb_.query('evil(x)')
```

Output:

```
Querying evil(x):
    1. evil(John)
```

Output:

```
            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()
```

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