VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence LAB

Submitted by

Puneeth Kumar HT (1BM22CS414)

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 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 "Artificial Intelligence lab" carried out by **Puneeth Kumar H T (1BM22CS414),** who is a 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 an **Artificial Intelligence lab (22CS5PCAIN)** work prescribed for the said degree.

M Lakshmi Neelima
Assistant Professor
Department of CSE
BMSCE, Bengaluru

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

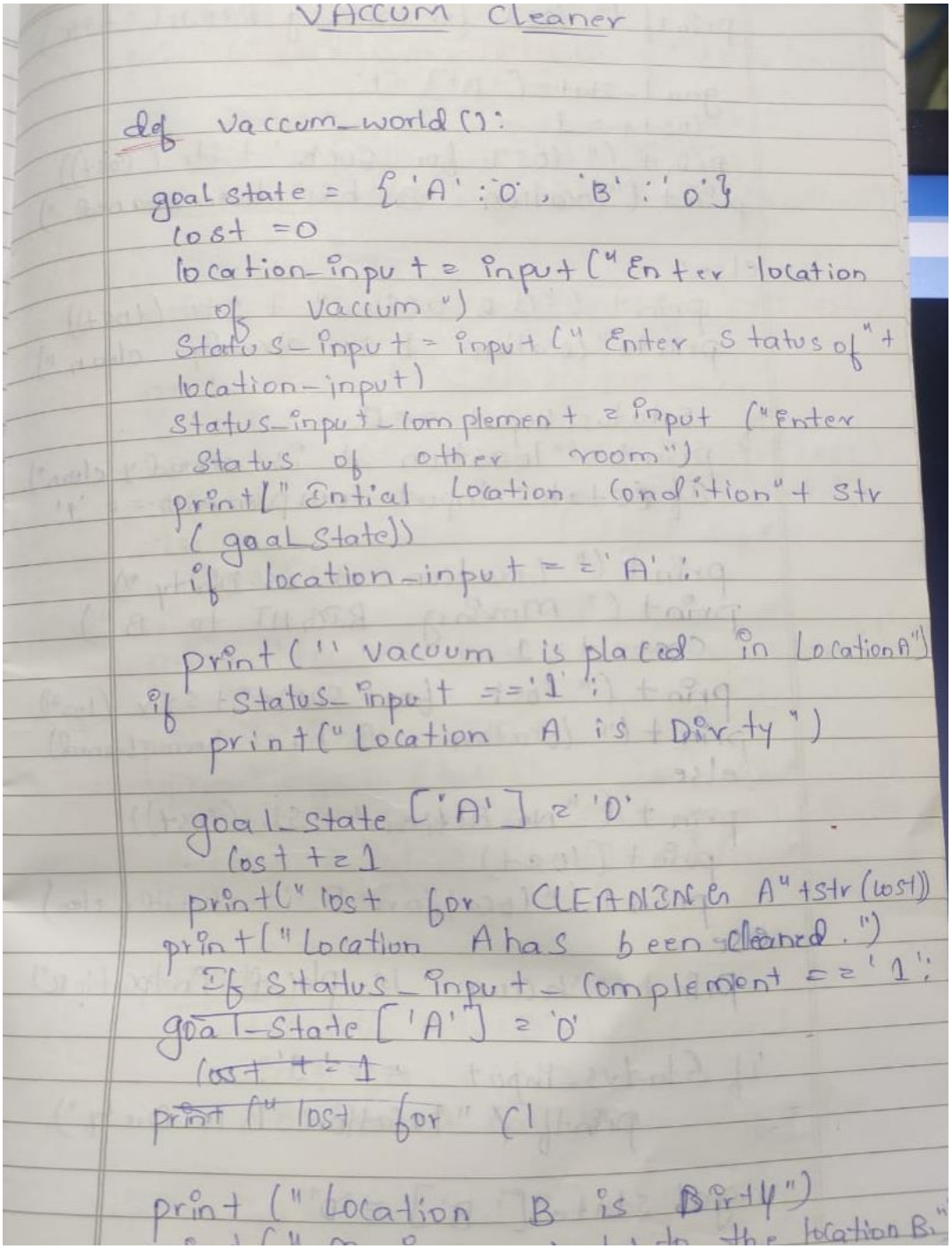
`

Program 1:
Implement the vacuum cleaner program(20/11/23)

```
def vacuum_world():
 # initializing goal_state
 # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0
    location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
    status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))
    if location_input == 'A':
    # Location A is Dirty.
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
     # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1 #cost for suck
            print("Cost for CLEANING A " + str(cost))
            print("Location A has been Cleaned.")
            if status_input_complement == '1':
    # if B is Dirty
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1 #cost for moving right
                print("COST for moving RIGHT" + str(cost))
                # suck the dirt and mark it as clean
                goal_state['B'] = '0'
                cost += 1 #cost for suck
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
            else:
                print("No action" + str(cost))
                # suck and mark clean
                print("Location B is already clean.")

        if status_input == '0':
            print("Location A is already clean ")
            if status_input_complement == '1':# if B is Dirty
                print("Location B is Dirty.")
                print("Moving RIGHT to the Location B. ")
                cost += 1 #cost for moving right
                print("COST for moving RIGHT " + str(cost))
```

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

```
        print("Location A has been Cleaned. ")
    else:
        print("No action " + str(cost))
        # suck and mark clean
        print("Location A is already clean.")
        # done cleaning
        print("GOAL STATE: ")
        print(goal_state)
        print("Performance Measurement: " + str(cost))

vacuum_world()
```



output

```
Enter Location of VacuumA
Enter status of A1
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to the Location B.
COST for moving RIGHT2
COST for SUCK 3
Location B has been Cleaned.
```

Program 2:

Implement the 8 Puzzle Bíeadth Ïiíst Seaích Algoíithm.(11/12/23)

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

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

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

    return temp  # Return the modified state

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

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

    pos_moves_it_can = []

    for i in d:
        pos_moves_it_can.append(gen(state, i, b))

    return [move_it_can for move_it_can in pos_moves_it_can if move_it_can not in
visited_states]

def bfs(src, target):
```

```
    queue = []
    queue.append(src)
    cost=0

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print("queue")
        for q in queue:
          print(q)
          print("..........***..........")

        print(source[0],'|',source[1],'|',source[2])
        print(source[3],'|',source[4],'|', source[5])
        print(source[6],'|', source[7],'|',source[8])
        print()
        cost=cost+1

        if source == target:
            print("success")
            print("path cost",cost)
            return

        poss_moves_to_do = possible_moves(source, exp)

        for move in poss_moves_to_do:
            if move not in exp and move not in queue:
                queue.append(move)

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

PAGE NO :
DATE :

PuzzleState

```
class PuzzleState:
    def __init__(self, board, parent = None, move = None):
        self.board = board
        self.parent = parent
        self.move = move

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self):
        return hash(str(board))

def print_solution(final_state):
    path = []
    while final_state:
        path.append(final_state)
        final_state = final_state.parent

    for t in reversed(Path):
        if t.move is not None:
            print("Move", t.move)
            print_board(t.board)

def print_board(board):
    for i in range(3):
        for j in range(3):
            print(board[i*3+j], end = " ")
        print()
```

Output

```
queue
1 | 2 | 3
4 | 5 | 6
0 | 7 | 8

queue
[1, 2, 3, 4, 5, 6, 7, 0, 8]
----------***------------
1 | 2 | 3
0 | 5 | 6
4 | 7 | 8

queue
[0, 2, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 0, 6, 4, 7, 8]
----------***------------
1 | 2 | 3
4 | 5 | 6
7 | 0 | 8
```

```
queue
[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]
----------***------------
0 | 2 | 3
1 | 5 | 6
4 | 7 | 8

queue
[1, 2, 3, 4, 0, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 4, 5, 6, 7, 8, 0]
----------***------------
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
1 | 2 | 3
5 | 0 | 6
4 | 7 | 8

queue
[1, 2, 3, 4, 5, 6, 7, 8, 0]
----------***------------
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 0, 3, 5, 2, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 7, 6, 4, 0, 8]
----------***------------
[1, 2, 3, 5, 6, 0, 4, 7, 8]
----------***------------
1 | 2 | 3
4 | 0 | 6
7 | 5 | 8

queue
[2, 0, 3, 1, 5, 6, 4, 7, 8]
----------***------------
[1, 0, 3, 5, 2, 6, 4, 7, 8]
----------***------------
[1, 2, 3, 5, 7, 6, 4, 0, 8]
----------***------------
[1, 2, 3, 5, 6, 0, 4, 7, 8]
----------***------------
[1, 0, 3, 4, 2, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 0, 4, 6, 7, 5, 8]
----------***------------
[1, 2, 3, 4, 6, 0, 7, 5, 8]
----------***------------
1 | 2 | 3
4 | 5 | 6
7 | 8 | 0

success
path cost 7
```

Program 3:

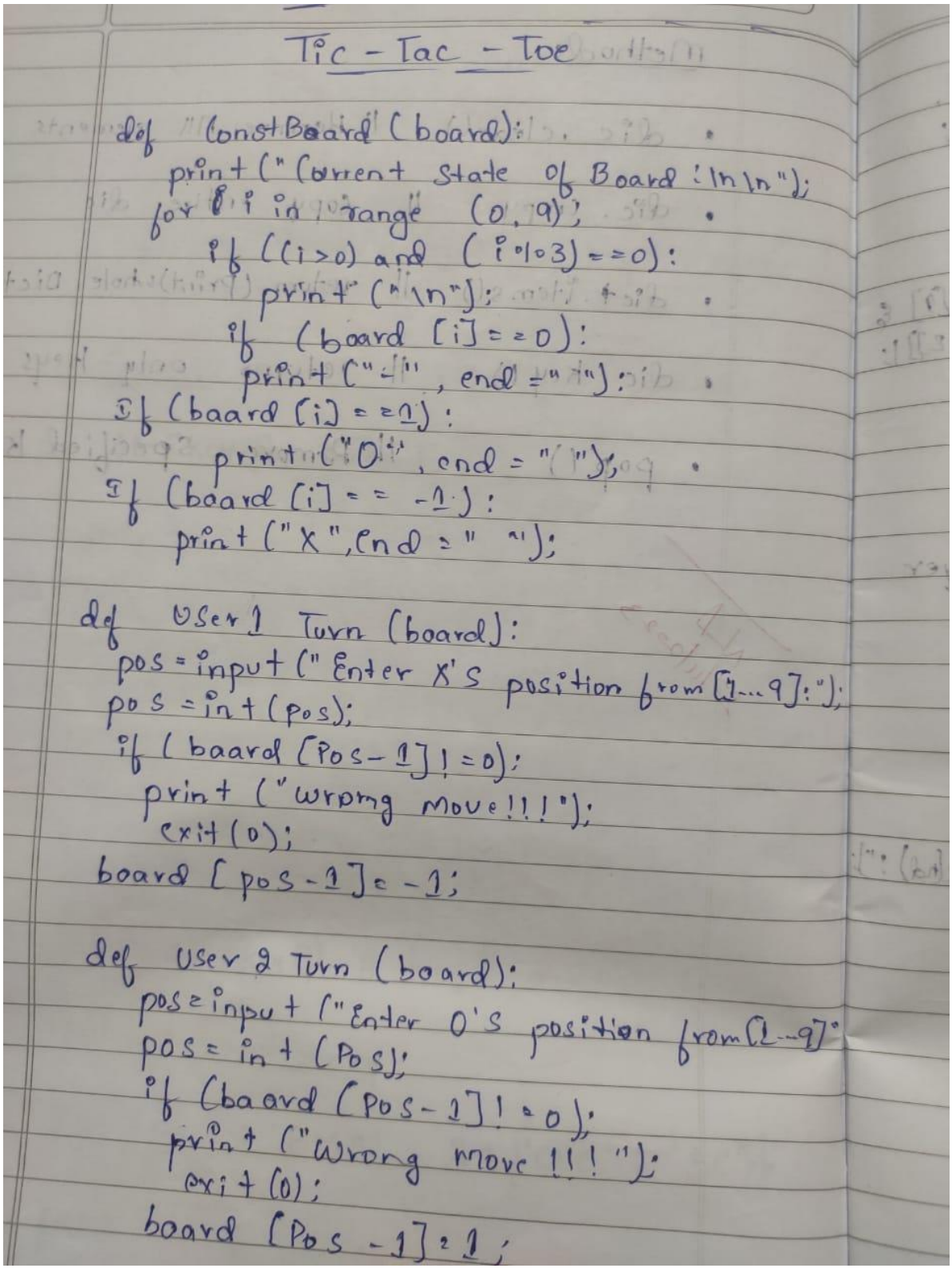
Exploíe the woíking of Tic Tac Toe using Min max stíategy. (11/12/23)

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

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

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

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

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

```
    for i in range(3):
      for j in range(3):
        if board[i][j] == " ":
            board[i][j] = "X"
            eval = minimax(depth + 1, True)
            board[i][j] = " "
            min_eval = min(min_eval, eval)

    return min_eval

def ai_move():
  best_move = None
  best_eval = float("-inf")
  for i in range(3):
    for j in range(3):
      if board[i][j] == " ":
        board[i][j] = "O"
        eval = minimax(0, False)
        board[i][j] = " "
        if eval > best_eval:
          best_eval = eval
          best_move = (i, j)

  return best_move

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

break

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



Tic - Tac - Toe

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

def User 1 Turn (board):
    pos = input ("Enter X's position from [1...9]:");
    pos = int (pos);
    if (baard [Pos-1] != 0);
        print ("wrong move!!!");
        exit (0);
    board [pos-1] = -1;

def User 2 Turn (board):
    pos = input ("Enter 0's position from [1...9]:
    pos = int (Pos);
    if (baard [pos-1] != 0);
        print ("Wrong move !!!");
        exit (0);
    baard [Pos-1] = 1;
```

output

```
0,0|0,1|0,2
1,0|1,1|1,2
2,0|,2,1|2,2


 | |
-----
 | |
-----
 | |
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 1
O| |
-----
 |X|
-----
 | |
-----
Enter row (0, 1, or 2): 0
Enter column (0, 1, or 2): 2
O| |X
-----
 |X|
-----
O| |
-----
Enter row (0, 1, or 2): 1
Enter column (0, 1, or 2): 0
O| |X
-----
X|X|O
-----
O| |
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 1
O|O|X
-----
X|X|O
-----
O|X|
-----
Enter row (0, 1, or 2): 2
Enter column (0, 1, or 2): 2
O|O|X
-----
X|X|O
-----
O|X|X
-----
It's a draw!
```

Program 4:
Implement Iteíative deepening seaích algoíithm.(18/12/23)

```
from collections import defaultdict

# This class represents a directed graph using adjacency
# list representation
class Graph:

    def__init__(self,vertices):

        # No. of vertices
        self.V = vertices

        # default dictionary to store graph
        self.graph = defaultdict(list)
        self.ans = list()

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    # A function to perform a Depth-Limited search
    # from given source 'src'
    def DLS(self,src,target,maxDepth,l):

        if src == target :
          #  print(self.ans)
            return True

        # If reached the maximum depth, stop recursing.
        if maxDepth <= 0 : return False

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1,l)):
                    l.append(i)
                    return True
        return False

    # IDDFS to search if target is reachable from v.
    # It uses recursive DLS()
```

```
    def IDDFS(self,src, target, maxDepth):

        # Repeatedly depth-limit search till the
        # maximum depth
        for i in range(maxDepth):
            l = []
            if (self.DLS(src, target, i,l)):
                l.append(src)
                l.reverse()
                return l
        return l

# Create a graph given in the above diagram
n,e = map(int ,input("Enter no.of vertices and edges").split())
g = Graph (n);
for i in range(e):
    a,b = map(int , input().split())
    g.addEdge(a,b)

# g.addEdge(0, 1)
# g.addEdge(0, 2)
# g.addEdge(1, 3)
# g.addEdge(1, 4)
# g.addEdge(2, 5)
# g.addEdge(2, 6)

target = int(input("Enter the target vertex"))
maxDepth = int(input("Enter the max depth"))
src = 0
l = g.IDDFS(src, target, maxDepth)
if len(l)!=0:
    print(l)
    print ("Target is reachable from source " +
        "within max depth")
else :
    print ("Target is NOT reachable from source " +
        "within max depth")
```

8/11/23

Iterative deepening

PAGE NO :
DATE :

```
def iterative deeping search (graph, start,
  goal):
depth_limit = 0
while true :
   result, path = depth limtal search (grooph
   , smart, goal, depth limit, [start])

if result == goal
    return result; path
     depth_limit += 1 ;

def depth_limited_search (graph, goal,
                          depth_limitpath):

  if (orent = goal
     return (orrent. path
  if depth_limit == 0;
     oeturn Now (1
  if depth_limit > 0;
   for weighor in graph [( ment]
  result, New_path = depth_limited_Search
                (graph Neighour, goal,
                  depth , limit - 1)
     if result == goal
        return result, Neu path

return None []
```

output

```
Enter no.of vertices and edges7 6
0 1
0 2
1 3
1 4
2 5
2 6
Enter the target vertex6
Enter the max depth3
[0, 2, 6]
Target is reachable from source within max depth
```

Program 5:

Implement A* foí 8 puzzle píoblem(8/1/24)

```
# Online Python compiler (interpreter) to run Python online.
# Write Python 3 code in this online editor and run it.
class Node:
    def__init_(self,data,level,fval):
        self.data = data
        self.level = level
        self.fval = fval

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

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


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

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


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

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

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

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


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

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        self.open.append(start)

        while True:
            cur = self.open[0]
```

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

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


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

A* 8 Puzzle

```
class Node:

    def __init__(self, data, level, fval):

        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):

        x, y = self.find(self.data, '_')

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

        return children

    def shuffle(self, puz, x1, y1, x2, y2):

        temp_puz = []
```

output

```
Enter the start state matrix

1 2 3
5 6 _
7 8 4
Enter the goal state matrix

1 2 3
5 8 6
_ 7 4
|
  |
 \'/
1 2 3
5 6 _
7 8 4
cost: 3

  |
  |
 \'/

1 2 3
5 _ 6
7 8 4
cost: 3

  |
  |
 \'/

1 2 3
5 8 6
7 _ 4
cost: 3

  |
  |
 \'/

1 2 3
```

Program 6:

Creation of Knowledge Base using prepositional logic and show that the query entails the KB or not (22/1/24)

```
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)]
variable={'p':0,'q':1, 'r':2}
kb=''
q=''
priority={'~':3,'v':1,'^':2}
def input_rules():
    global kb, y
    kb = (input("Enter rule: "))
    y = input("Enter the Query: ")
def entailment():
    global kb, q
    print('*'*10+"Truth Table Reference"+'*'*10)
    print('kb','alpha')
    print('*'*10)
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(y), comb)
        print(s, f)
        print('-'*10)
        if s and not f:
            return False
    return  True
def isOperand(c):
    return c.isalpha() and c!='v'

def isLeftParanthesis(c):
    return c == '('

def isRightParanthesis(c):
    return c == ')'

def isEmpty(stack):
    return len(stack) == 0
```

```
def peek(stack):
    return stack[-1]

def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1]<=priority[c2]
    except KeyError:
        return False
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()

    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i,val2,val1))
    return stack.pop()
def _eval(i, val1, val2):
```

```
    if i == '^':
        return val2 and val1
    return val2 or val1

input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

output:

```
Enter rule: (pvq)^(p^~r)
Enter the Query: p
**********Truth Table Reference**********
kb alpha
**********
False True
----------
True True
----------
False True
----------
True True
----------
False False
----------
False False
----------
False False
----------
False False
----------
The Knowledge Base entails query
>
```

KnowledgeBase Entailment

variable {"P":0, "q"} "r":r}
priority = {"~" 3 "v"} "^":2}

def val (i, val 1, val 2);
  if i == "^"
    return val 2 and val 1
  return val 2 (or) val 2

def is operand (c):
  return c, is alpha () and c1 = "v"
def is left Paranthesis (c):
  return c == "("

def is Empty (stack):
  return len (stack) == 0
  def peek (stack)
    return stack[-1]
def has less (or) Equal Priorty (c1, c2)
  try
  return priority (c1) <= prority (c2)
    except key Error
      return false
def to Postfix (infix)
  stack = []
  Postfix = " "

Program 7:

Creation of Knowledge Base using prepositional logic and prove the query using resolution(22/1/24)

```
kb = []


def CLEAR():
    global kb
    kb = []


def TELL(sentence):
    global kb
    # If the sentence is a clause, insert directly.
    if isClause(sentence):
        kb.append(sentence)
    # If not, convert to CNF, and then insert clauses one by one.
    else:
        sentenceCNF = convertCNF(sentence)
        if not sentenceCNF:
            print("Illegal input")
            return
        # Insert clauses one by one when there are multiple clauses
        if isAndList(sentenceCNF):
            for s in sentenceCNF[1:]:
                kb.append(s)
        else:
            kb.append(sentenceCNF)


def ASK(sentence):
    global kb

    # Negate the sentence, and convert it to CNF accordingly.
    if isClause(sentence):
        neg = negation(sentence)
    else:
        sentenceCNF = convertCNF(sentence)
```

```
    if not sentenceCNF:
        print("Illegal input")
        return
    neg = convertCNF(negation(sentenceCNF))

# Insert individual clauses that we need to ask to ask_list.
ask_list = []
if isAndList(neg):
    for n in neg[1:]:
        nCNF = makeCNF(n)
        if type(nCNF).__name__ == 'list':
            ask_list.insert(0, nCNF)
        else:
            ask_list.insert(0, nCNF)
else:
    ask_list = [neg]
clauses = ask_list + kb[:]
while True:
    new_clauses = []
    for c1 in clauses:
        for c2 in clauses:
            if c1 is not c2:
                resolved = resolve(c1, c2)
                if resolved == False:
                    continue
                if resolved == []:
                    return True
                new_clauses.append(resolved)

    if len(new_clauses) == 0:
        return False

    new_in_clauses = True
    for n in new_clauses:
        if n not in clauses:
            new_in_clauses = False
            clauses.append(n)

    if new_in_clauses:
        return False
```

```
    return False


def resolve(arg_one, arg_two):
    resolved = False

    s1 = make_sentence(arg_one)
    s2 = make_sentence(arg_two)

    resolve_s1 = None
    resolve_s2 = None

    # Two for loops that iterate through the two clauses.
    for i in s1:
        if isNotList(i):
            a1 = i[1]
            a1_not = True
        else:
            a1 = i
            a1_not = False

        for j in s2:
            if isNotList(j):
                a2 = j[1]
                a2_not = True
            else:
                a2 = j
                a2_not = False

            # cancel out two literals such as 'a' $ ['not', 'a']
            if a1 == a2:
                if a1_not != a2_not:
                    # Return False if resolution already happend
                    # but contradiction still exists.
                    if resolved:
                        return False
                    else:
                        resolved = True
                        resolve_s1 = i
                        resolve_s2 = j
```

```
                    break
                # Return False if not resolution happened
    if not resolved:
        return False

    # Remove the literals that are canceled
    s1.remove(resolve_s1)
    s2.remove(resolve_s2)

    # # Remove duplicates
    result = clear_duplicate(s1 + s2)

    # Format the result.
    if len(result) == 1:
        return result[0]
    elif len(result) > 1:
        result.insert(0, 'or')

    return result


def make_sentence(arg):
    if isLiteral(arg) or isNotList(arg):
        return [arg]
    if isOrList(arg):
        return clear_duplicate(arg[1:])
    return


def clear_duplicate(arg):
    result = []
    for i in range(0, len(arg)):
        if arg[i] not in arg[i+1:]:
            result.append(arg[i])
    return result


def isClause(sentence):
    if isLiteral(sentence):
        return True
```

```
    if isNotList(sentence):
        if isLiteral(sentence[1]):
            return True
        else:
            return False
    if isOrList(sentence):
        for i in range(1, len(sentence)):
            if len(sentence[i]) > 2:
                return False
            elif not isClause(sentence[i]):
                return False
        return True
    return False


def isCNF(sentence):
    if isClause(sentence):
        return True
    elif isAndList(sentence):
        for s in sentence[1:]:
            if not isClause(s):
                return False
        return True
    return False


def negation(sentence):
    if isLiteral(sentence):
        return ['not', sentence]
    if isNotList(sentence):
        return sentence[1]

    # DeMorgan:
    if isAndList(sentence):
        result = ['or']
        for i in sentence[1:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', sentence])
```

```
        return result
    if isOrList(sentence):
        result = ['and']
        for i in sentence[:]:
            if isNotList(sentence):
                result.append(i[1])
            else:
                result.append(['not', i])
        return result
    return None


def convertCNF(sentence):
    while not isCNF(sentence):
        if sentence is None:
            return None
        sentence = makeCNF(sentence)
    return sentence


def makeCNF(sentence):
    if isLiteral(sentence):
        return sentence

    if (type(sentence).__name__== 'list'):
        operand = sentence[0]
        if isNotList(sentence):
            if isLiteral(sentence[1]):
                return sentence
            cnf = makeCNF(sentence[1])
            if cnf[0] == 'not':
                return makeCNF(cnf[1])
            if cnf[0] == 'or':
                result = ['and']
                for i in range(1, len(cnf)):
                    result.append(makeCNF(['not', cnf[i]]))
                return result
            if cnf[0] == 'and':
                result = ['or']
                for i in range(1, len(cnf)):
```

```
            result.append(makeCNF(['not', cnf[i]]))
        return result
    return "False: not"

if operand == 'implies' and len(sentence) == 3:
    return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])

if operand == 'biconditional' and len(sentence) == 3:
    s1 = makeCNF(['implies', sentence[1], sentence[2]])
    s2 = makeCNF(['implies', sentence[2], sentence[1]])
    return makeCNF(['and', s1, s2])

if isAndList(sentence):
    result = ['and']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isAndList(cnf):
            for i in range(1, len(cnf)):
                result.append(makeCNF(cnf[i]))
            continue
        result.append(makeCNF(cnf))
    return result

if isOrList(sentence):
    result1 = ['or']
    for i in range(1, len(sentence)):
        cnf = makeCNF(sentence[i])
        # Distributivity:
        if isOrList(cnf):
            for i in range(1, len(cnf)):
                result1.append(makeCNF(cnf[i]))
            continue
        result1.append(makeCNF(cnf))
        # Associativity:
    while True:
        result2 = ['and']
        and_clause = None
        for r in result1:
            if isAndList(r):
```

```
                    and_clause = r
                    break

            # Finish when there's no more 'and' lists
            # inside of 'or' lists
            if not and_clause:
                return result1

            result1.remove(and_clause)

            for i in range(1, len(and_clause)):
                temp = ['or', and_clause[i]]
                for o in result1[1:]:
                    temp.append(makeCNF(o))
                result2.append(makeCNF(temp))
            result1 = makeCNF(result2)
        return None
    return None


def isLiteral(item):
    if type(item).__name__ == 'str':
        return True
    return False


def isNotList(item):
    if type(item).__name__ == 'list':
        if len(item) == 2:
            if item[0] == 'not':
                return True
    return False


def isAndList(item):
    if type(item).__name__ == 'list':
        if len(item) > 2:
            if item[0] == 'and':
                return True
    return False
```

```
def isOrList(item):
    if type(item).__name__== 'list':
        if len(item) > 2:
            if item[0] == 'or':
                return True
    return False


CLEAR()

TELL('p')
TELL(['implies', ['and', 'p', 'q'], 'r'])
TELL(['implies', ['or', 's', 't'], 'q'])
TELL('t')
print(ASK('r'))
```

Output

(Q) Create a knowledge base using proposition logic and prove the query using resolution.

```
# import re

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

def negate(term):
    return f'~{term}' if len(term) == 1 and 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 = r'(~*[PQRS])'
    terms = re.findall(exp, rule)
    return terms
```

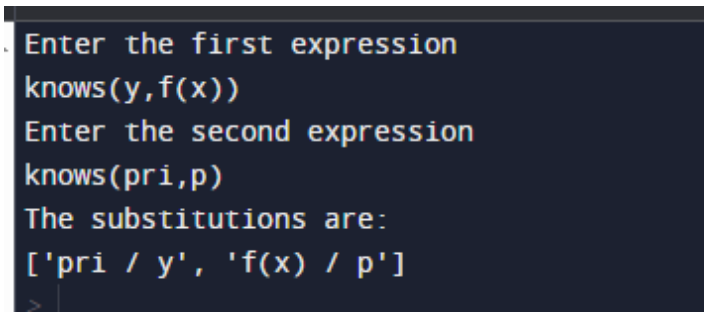
Program 8:

Implement unification in first order logic(29/1/24)

```
import re
def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

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)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    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:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
            return []

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

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

    if isVariable(exp1):
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    if isVariable(exp2):
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot
be unified")
        return []

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

```
    initialSubstitution = unify(head1, head2)
    if not initialSubstitution:
        return []
    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 []

    return initialSubstitution + remainingSubstitution
def main():
    print("Enter the first expression")
    e1 = input()
    print("Enter the second expression")
    e2 = input()
    substitutions = unify(e1, e2)
    print("The substitutions are:")
    print([' / '.join(substitution) for substitution in substitutions])
main()
```



Unification of first order

```
def unify (expr1, expr2):

    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    if func1 != func2

        print ("Expressions cannot be unified. Different functions")

        return None

    args1 = args1.rstrip(')').split(",")
    args2 = args2.rstrip(')').split(',')

    substitution = {}

    for a1, a2 in zip (args1, args2)

        if a1.islower() and a2.islower() & a1 != a
            substitution [a1] = a2
        elif a1.islower() & not a2.islower():
            substitution [a1] = a2
        elif Not a1.islower() & a2.islower():
            substitution [a2] = a1
        elif a1 != a2
```

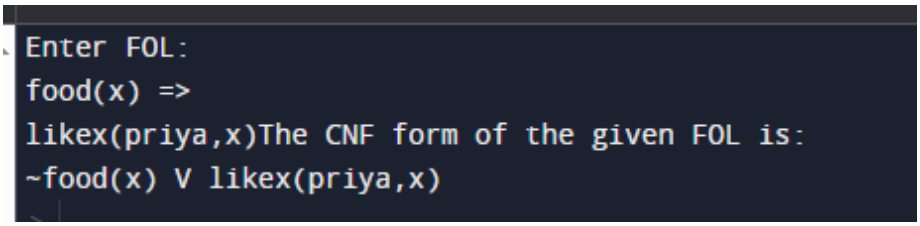
Program 9:

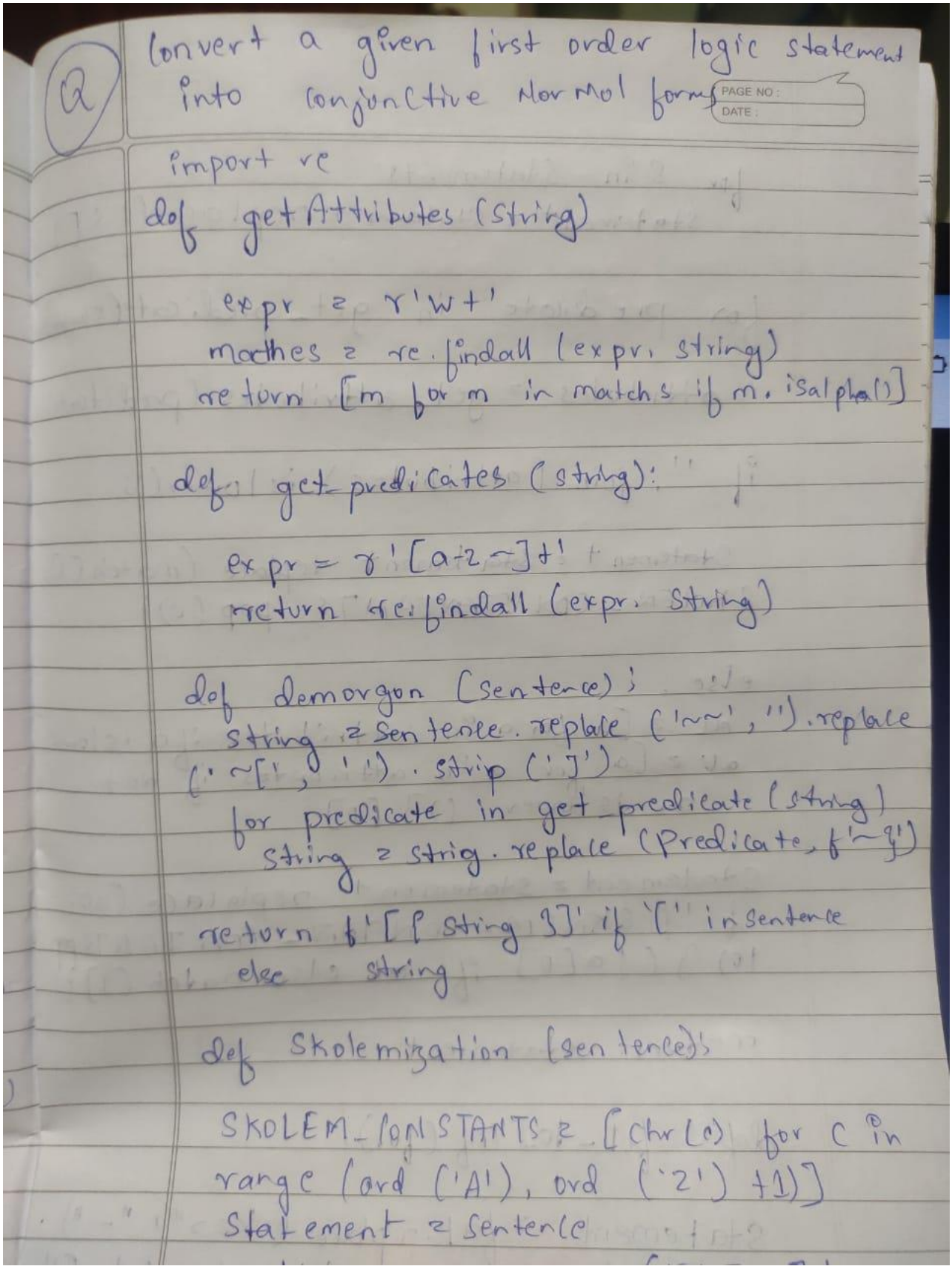
Convert a given first order logic statement into Conjunctive Normal Form (CNF).(29/1/24)

```
import re

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

def  getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)
def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'
    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
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] + 'V' + 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 = '(~[∀V∃].)'
    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
def main():
    print("Enter FOL:")
    fol = input()
    print("The CNF form of the given FOL is: ")
    print(Skolemization(fol_to_cnf(fol)))
main()
```





Q) Convert a given first order logic statement into conjunctive Normal form

PAGE NO :
DATE :

```
import re
def getAttributes (string)
    expr = r'\w+'
    matches = re.findall (expr, string)
    return [m for m in matchs if m.isalpha()]

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

def demorgon (sentence):
    string = Sentence.replace ('~~', '').replace
('~[', '[').strip (']')
    for predicate in get_predicate (string)
        string = strig.replace (Predicate, f'~{}')

return f'[{string}]' if '[' in sentence
    else string

def Skolemization (sentence):
    SKOLEM_CONSTANTS = [chr(c) for c in
range (ord ('A'), ord ('Z') +1)]
    Statement = Sentence
```

Program 10:

Forward Chaining (29/1/24)

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}')
def main():
    kb = KB()
    print("Enter KB: (enter e to exit)")
    while True:
        t = input()
        if(t == 'e'):
            break
        kb.tell(t)
    print("Enter Query:")
    q = input()
    kb.query(q)
```

```
    kb.display()
main()
```

Output

```
Enter KB: (enter e to exit)
missile(x) => weapon(x)
missile(m1)
enemy(x,america) => hostile(x)
american(west)
enemy(china,america)
owns(china,m1)
missile(x) & owns(china,x) => sells(west,x,china)
american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)
e
Enter Query:
criminal(x)
Querying criminal(x):
    1. criminal(west)
All facts:
    1. weapon(m1)
    2. criminal(west)
    3. missile(m1)
    4. owns(china,m1)
    5. american(west)
    6. sells(west,m1,china)
    7. hostile(china)
    8. enemy(china,america)
>
```

(Q) Create a knowledge base consisting of first order logic statements & prove the given query using forward reasoning

```
import re

def is variable (x):
    return len(x) == 1 & x.islower() & x.isalp

def get Attributes (string)
    expr = r'
    matches = r.findall (expr, string
    return matches

def get Predicates (string):
    expr = '([a-z~]+) [^&|]+
    retur re findall (exp, string)

class fact:

    def - init - (self, expression)

    self.expression = expression
    predicate, params = self.split Exp - (exp)
    Self.predicate = predicate
```