B.M.S. COLLEGE OF ENGINEERING BENGALURU
Autonomous Institute, Affiliated to VTU

Lab Record

Artificial Intelligence

Submitted in partial fulfillment for the 5th Semester Laboratory

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

HEMAMALA MN (1BM20CS056)

Department of Computer Science and Engineering
B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560 019
OCT-FEB 2023

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (20CS5PCAIP) laboratory has been carried out by HEMAMALA MN (1BM20CS056) during the 5 th Semester October-January-2023.

Signature of the Faculty Incharge:

Prof.Sunayana
Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

Program 1: Implement Tic –Tac –Toe Game.

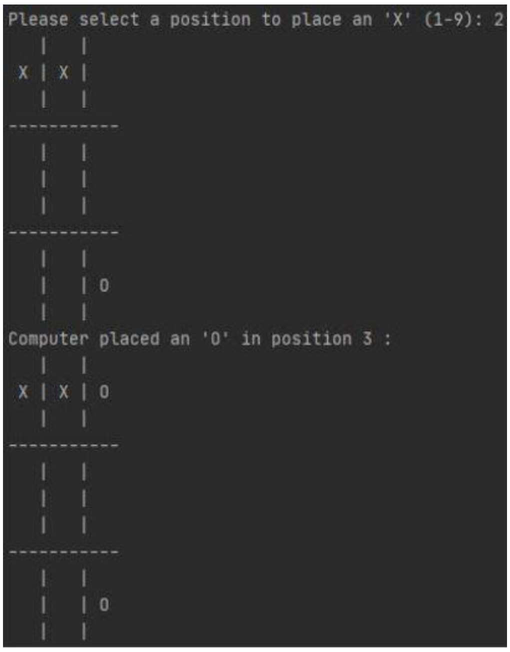
```
board = ['
' for x in
range(10)]


def insertLetter(letter, pos):
    board[pos] = letter


def spaceIsFree(pos):
    return board[pos] == ' '


def printBoard(board):
    print('   |   |')
    print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])
    print('   |   |')
    print('-----------')
    print('   |   |')
    print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('   |   |')


def isWinner(bo, le):
    return (bo[7] == le and bo[8] == le and bo[9] == le) or (bo[4] == le and
bo[5] == le and bo[6] == le) or (
                bo[1] == le and bo[2] == le and bo[3] == le) or (bo[1] == le and
bo[4] == le and bo[7] == le) or (
                       bo[2] == le and bo[5] == le and bo[8] == le) or (
                       bo[3] == le and bo[6] == le and bo[9] == le) or (
                       bo[1] == le and bo[5] == le and bo[9] == le) or (bo[3] ==
le and bo[5] == le and bo[7] == le)


def playerMove():
```

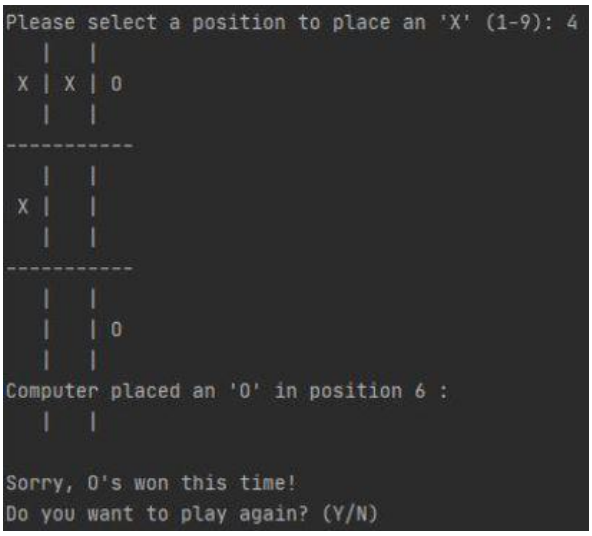
```
    run = True
    while run:
        move = input('Please select a position to place an \'X\' (1-9): ')
        try:
            move = int(move)
            if move > 0 and move < 10:
                if spaceIsFree(move):
                    run = False
                    insertLetter('X', move)
                else:
                    print('Sorry, this space is occupied!')
            else:
                print('Please type a number within the range!')
        except:
            print('Please type a number!')


def compMove():
    possibleMoves = [x for x, letter in enumerate(board) if letter == ' ' and x
!= 0]
    move = 0

    for let in ['O', 'X']:
        for i in possibleMoves:
            boardCopy = board[:]
            boardCopy[i] = let
            if isWinner(boardCopy, let):
                move = i
                return move

    cornersOpen = []
    for i in possibleMoves:
        if i in [1, 3, 7, 9]:
            cornersOpen.append(i)

    if len(cornersOpen) > 0:
        move = selectRandom(cornersOpen)
        return move

    if 5 in possibleMoves:
        move = 5
        return move

    edgesOpen = []
    for i in possibleMoves:
        if i in [2, 4, 6, 8]:
```

```
            edgesOpen.append(i)

    if len(edgesOpen) > 0:
        move = selectRandom(edgesOpen)

    return move


def selectRandom(li):
    import random
    ln = len(li)
    r = random.randrange(0, ln)
    return li[r]


def isBoardFull(board):
    if board.count(' ') > 1:
        return False
    else:
        return True


def main():
    print('Welcome to Tic Tac Toe!')
    printBoard(board)

    while not (isBoardFull(board)):
        if not (isWinner(board, 'O')):
            playerMove()
            printBoard(board)
        else:
            print('Sorry, O\'s won this time!')
            break

        if not (isWinner(board, 'X')):
            move = compMove()
            if move == 0:
                print('Tie Game!')
            else:
                insertLetter('O', move)
                print('Computer placed an \'O\' in position', move, ':')
                printBoard(board)
        else:
            print('X\'s won this time! Good Job!')
            break
```

```
        if isBoardFull(board):
            print('Tie Game!')



    while True:
        answer = input('Do you want to play again? (Y/N)')
        if answer.lower() == 'y' or answer.lower == 'yes':
            board = [' ' for x in range(10)]
            print('-----------------------------------')
            main()
        else:
            break
```

Output:





Program 2: Solve 8 puzzle problem using bfs.

```
def bfs(src,target):
    queue = []
    queue.append(src)

    exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)

        print(source)

        if source==target:
            print("success")
            return

        poss_moves_to_do = []
        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)

def possible_moves(state,visited_states):
    #index of empty spot
    b = state.index(-1)

    #directions array
    d = []
    #Add all the possible directions

    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')


    # If direction is possible then add state to move
    pos_moves_it_can = []

    # for all possible directions find the state if that move is played
```

```python
    ### Jump to gen function to generate all possible moves in the given
directions


    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 new state with tested move to later check if "src == target"
    return temp
```

Output:

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

```
[1, 2, 3, -1, 4, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, -1, 7, 8]
[-1, 2, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 4, -1, 5, 6, 7, 8]
[1, 2, 3, 6, 4, 5, 7, -1, 8]
[2, -1, 3, 1, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 7, 5, 6, -1, 8]
[1, -1, 3, 4, 2, 5, 6, 7, 8]
[1, 2, 3, 4, 5, -1, 6, 7, 8]
success
```

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

```
[2, -1, 3, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, -1, 4, 7, 6, 5]
[-1, 2, 3, 1, 8, 4, 7, 6, 5]
[2, 3, -1, 1, 8, 4, 7, 6, 5]
[2, 8, 3, 1, 6, 4, 7, -1, 5]
[2, 8, 3, -1, 1, 4, 7, 6, 5]
[2, 8, 3, 1, 4, -1, 7, 6, 5]
[1, 2, 3, -1, 8, 4, 7, 6, 5]
[2, 3, 4, 1, 8, -1, 7, 6, 5]
[2, 8, 3, 1, 6, 4, -1, 7, 5]
[2, 8, 3, 1, 6, 4, 7, 5, -1]
[2, 8, 3, 7, 1, 4, -1, 6, 5]
[-1, 8, 3, 2, 1, 4, 7, 6, 5]
[2, 8, 3, 1, 4, 5, 7, 6, -1]
[2, 8, -1, 1, 4, 3, 7, 6, 5]
[1, 2, 3, 7, 8, 4, -1, 6, 5]
[1, 2, 3, 8, -1, 4, 7, 6, 5]
success
```

Program 3: Solve 8 puzzle problem using iddfs.

DATE:31/03/2021

```
def dfs(src,target,limit,visited_states):
   if src == target:
      return True
   if limit <= 0:
      return False
   visited_states.append(src)
   moves = possible_moves(src,visited_states)
   for move in moves:
      if dfs(move, target, limit-1, visited_states):
         return True
   return False

def possible_moves(state,visited_states):
   b = state.index(-1)
   d = []
   if b not in [0,1,2]:
      d += 'u'
   if b not in [6,7,8]:
      d += 'd'
   if b not in [2,5,8]:
      d += 'r'
   if b not in [0,3,6]:
      d += 'l'
   pos_moves = []
   for move in d:
      pos_moves.append(gen(state,move,b))
   return [move for move in pos_moves if move not in visited_states]

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

def iddfs(src,target,depth):
   for i in range(depth):
```

```
        visited_states = []
        if dfs(src,target,i+1,visited_states):
            return True
    return False
```

Output:

```
In [5]: #Test 1
        src = [1,2,3,-1,4,5,6,7,8]
        target = [1,2,3,4,5,-1,6,7,8]

        depth = 1
        iddfs(src, target, depth)

Out[5]: False

In [6]: #Test 2
        src = [3,5,2,8,7,6,4,1,-1]
        target = [-1,3,7,8,1,5,4,6,2]

        depth = 1
        iddfs(src, target, depth)

Out[6]: False

In [7]: # Test 2
        src = [1,2,3,-1,4,5,6,7,8]
        target=[1,2,3,6,4,5,-1,7,8]

        depth = 1
        iddfs(src, target, depth)

Out[7]: True
```

Program 4: Solve 8 puzzle problem using A*.

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

Output:

```
puz = Puzzle(3)
puz.process()
Enter the start state matrix

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

1 2 3
8 _ 4
7 6 5



  |
  |
 \'/

2 _ 3
1 8 4
7 6 5

  |
  |
 \'/

_ 2 3
1 8 4
7 6 5

  |
  |
 \'/

1 2 3
_ 8 4
7 6 5

  |
  |
 \'/

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

Program 5: Implement vacuum cleaner agent.

```
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
```

```python
            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.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
```

```
Enter Location of Vacuum A
Enter status of  A0
Enter status of other room0
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean.
No action 0
Location A is already clean.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 0
```

```
Enter Location of VacuumB
Enter status of B0
Enter status of other room1
Initial Location Condition{'A': '0', 'B': '0'}
Vacuum is placed in location B
0
Location B is already clean.
Location A is Dirty.
Moving LEFT to the Location A.
COST for moving LEFT 1
Cost for SUCK 2
Location A has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

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

```
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, q
    kb = (input("Enter rule: "))
    q = 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(q), 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
```

Output:

```
#Test 1
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

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

```
#Test 2
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails query")
else:
    print("The Knowledge Base does not entail query")
```

```
Enter rule: (pvq)^(~rvp)
Enter the Query: p^r
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True False
----------
The Knowledge Base does not entail query
```

Program 7: Create a knowledgebase using prepositional logic and prove the given query using resolution.

```python
import re

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

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

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

def contradiction(query, clause):
    contradictions = [ f'{query}v{negate(query)}', f'{negate(query)}v{query}']
    return clause in contradictions or reverse(clause) in contradictions

def resolve(kb, query):
    temp = kb.copy()
    temp += [negate(query)]
    steps = dict()
    for rule in temp:
        steps[rule] = 'Given.'
    steps[negate(query)] = 'Negated conclusion.'
    i = 0
    while i < len(temp):
        n = len(temp)
        j = (i + 1) % n
        clauses = []
        while j != i:
            terms1 = split_terms(temp[i])
            terms2 = split_terms(temp[j])
            for c in terms1:
                if negate(c) in terms2:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate(c)]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen[1]):
                            clauses += [f'{gen[0]}v{gen[1]}']
                        else:
                            if contradiction(query,f'{gen[0]}v{gen[1]}'):
                                temp.append(f'{gen[0]}v{gen[1]}')
```

```
                                steps[''] = f"Resolved {temp[i]} and {temp[j]}
to {temp[-1]}, which is in turn null. \
                                \nA contradiction is found when

{negate(query)} is assumed as true. Hence, {query} is true."
                                return steps
                        elif len(gen) == 1:
                            clauses += [f'{gen[0]}']
                        else:
                            if contradiction(query,f'{terms1[0]}v{terms2[0]}'):
                                temp.append(f'{terms1[0]}v{terms2[0]}')
                                steps[''] = f"Resolved {temp[i]} and {temp[j]} to
{temp[-1]}, which is in turn null. \
                                \nA contradiction is found when {negate(query)} is
assumed as true. Hence, {query} is true."
                                return steps
                for clause in clauses:
                    if clause not in temp and clause != reverse(clause) and
reverse(clause) not in temp:
                        temp.append(clause)
                        steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'
                j = (j + 1) % n
            i += 1
    return steps

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

def main():
    print("Enter the kb:")
    kb = input()
    print("Enter the query:")
    query = input()
    resolution(kb,query)
```

Output:

```
#test 1
#(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
main()
```

```
Enter the kb:
Rv~P Rv~Q ~RvP ~RvQ
Enter the query:
R

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

```
#test 2
#(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
main()
```

```
Enter the kb:
PvQ PvR ~PvR RvS Rv~Q ~Sv~Q
Enter the query:
R

Step    |Clause |Derivation
------------------------------
 1.     | PvQ   | Given.
 2.     | PvR   | Given.
 3.     | ~PvR  | Given.
 4.     | RvS   | Given.
 5.     | Rv~Q  | Given.
 6.     | ~Sv~Q | Given.
 7.     | ~R    | Negated conclusion.
 8.     | QvR   | Resolved from PvQ and ~PvR.
 9.     | Pv~S  | Resolved from PvQ and ~Sv~Q.
 10.    | P     | Resolved from PvR and ~R.
 11.    | ~P    | Resolved from ~PvR and ~R.
 12.    | Rv~S  | Resolved from ~PvR and Pv~S.
 13.    | R     | Resolved from ~PvR and P.
 14.    | S     | Resolved from RvS and ~R.
 15.    | ~Q    | Resolved from Rv~Q and ~R.
 16.    | Q     | Resolved from ~R and QvR.
 17.    | ~S    | Resolved from ~R and Rv~S.
 18.    |       | Resolved ~R and R to ~RvR, which is in turn null.
A contradiction is found when ~R is assumed as true. Hence, R is true.
```

Program 8: Implement unification in first order logic

```python
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)
```

```python
    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)
```

```python
        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])
```

Output:

```
main()
```

```
Enter the first expression
knows(f(x),y)
Enter the second expression
knows(J,John)
The substitutions are:
['J / f(x)', 'John / y']
```

```
main()
```

```
Enter the first expression
Student(x)
Enter the second expression
Teacher(Rose)
Cannot be unified as the predicates do not match!
The substitutions are:
[]
```

```
main()
```

```
Enter the first expression
knows(John,x)
Enter the second expression
knows(y,Mother(y))
The substitutions are:
['John / y', 'Mother(y) / x']
```

```
main()
```

```
Enter the first expression
like(A,y)
Enter the second expression
like(K,g(x))
A and K are constants. Cannot be unified
The substitutions are:
[]
```

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

```
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
```

Output

```
#Test 1
main()
```

```
Enter FOL:
∀x food(x) => likes(John, x)
The CNF form of the given FOL is:
~ food(A) V likes(John, A)
```

```
#Test 2
main()
```

```
Enter FOL:
∀x[∃z[loves(x,z)]]
The CNF form of the given FOL is:
[loves(x,B(x))]
```

```
#Test 3
main()
```

```
Enter FOL:
[american(x)^weapon(y)^sells(x,y,z)^hostile(z)] => criminal(x)
The CNF form of the given FOL is:
[~american(x)V~weapon(y)V~sells(x,y,z)V~hostile(z)] V criminal(x)
```

Program 10: Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.

```
import re

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

def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

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

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

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

    def getResult(self):
        return self.result

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

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

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

class Implication:
```

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

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

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

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

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

```python
    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()
```

Output:

```
main()
```

```
Enter KB: (enter e to exit)
missile(x)=>weapon(x)
missile(M1)
enemy(x,America)=>hostile(x)
american(West)
enemy(Nono,America)
owns(Nono,M1)
missile(x)&owns(Nono,x)=>sells(West,x,Nono)
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. missile(M1)
        2. sells(West,M1,Nono)
        3. hostile(Nono)
        4. owns(Nono,M1)
        5. weapon(M1)
        6. criminal(West)
        7. american(West)
        8. enemy(Nono,America)
```