AI Lab Report

Submitted by

ANIKA SINGH**(1BM20CS014)**
Batch: A1

Course: Artificial Intelligence
Course Code: 20CS5PCAIP
Sem & Section: 5TH A

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

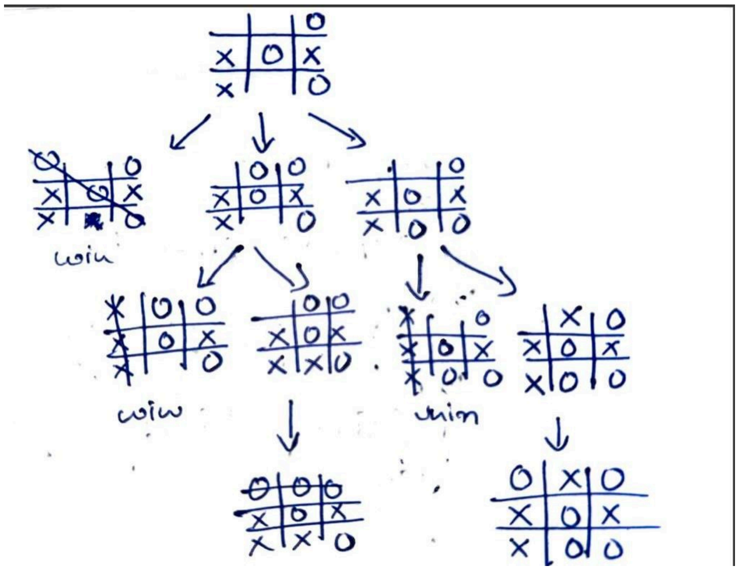
B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2022-2023

Lab-Program-1

Implement Tic –Tac –Toe Game.

Objective: The objective of tic-tac-toe is that players has to position their marks so that they make a continuous lines of three cells horizontally, vertically or diagonally.

Flowchart/State Space Diagram:



Code:

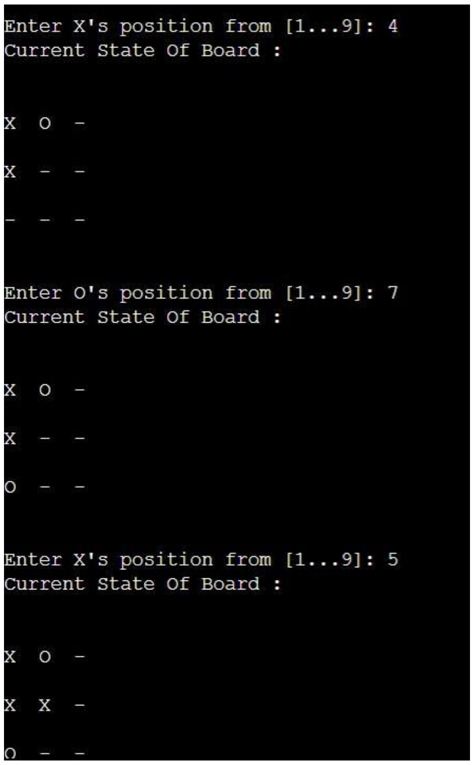
```
def ConstBoard(board):
    print("Current State Of Board : \n\n");
    for i in range (0,9):
        if((i>0) and (i%3)==0):
            print("\n");
        if(board[i]==0):
            print("- ",end=" ");
        if (board[i]==1):
            print("O ",end=" ");
        if(board[i]==-1):
   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

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

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

```
def minimax(board,player):

    x=analyzeboard(board);

    if(x!=0):

        return (x*player);

    pos=-1;

    value=-2;

    for i in range(0,9):

        if(board[i]==0):

            board[i]=player;

            score=-minimax(board,(player*-1));

            if(score>value):

                value=score;

                pos=i;

            board[i]=0;


    if(pos==-1):

        return 0;

    return value;


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

    board[pos]=1;



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

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


def main():
    choice=input("Enter 1 for single player, 2 for multiplayer: ");
    choice=int(choice);
    #The broad is considered in the form of a single dimentional array.
    #One player moves 1 and other move -1.
    board=[0,0,0,0,0,0,0,0,0];
    if(choice==1):
        print("Computer : O Vs. You : X");
        player= input("Enter to play 1(st) or 2(nd) :");
        player = int(player);
        for i in range (0,9):
            if(analyzeboard(board)!=0):
                break;
            if((i+player)%2==0):
                CompTurn(board);
            else:
                ConstBoard(board);
                User1Turn(board);
```

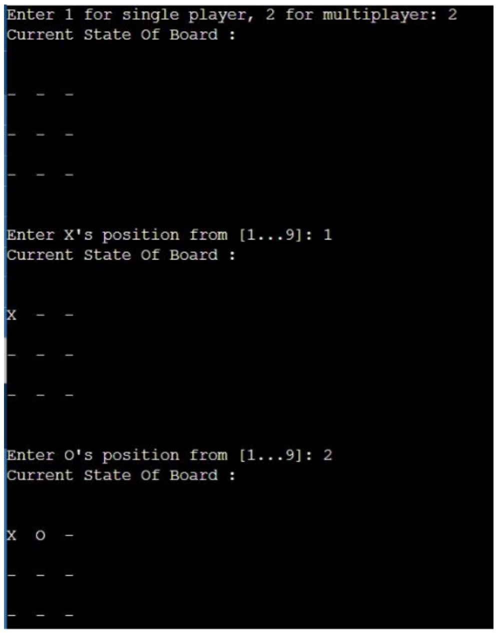
```
else:
    for i in range (0,9):
        if(analyzeboard(board)!=0):
            break;
        if((i)%2==0):
            ConstBoard(board);
            User1Turn(board);
        else:
            ConstBoard(board);
            User2Turn(board);



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

```
        print("X Loose!!! O Wins !!!!")



main()
```

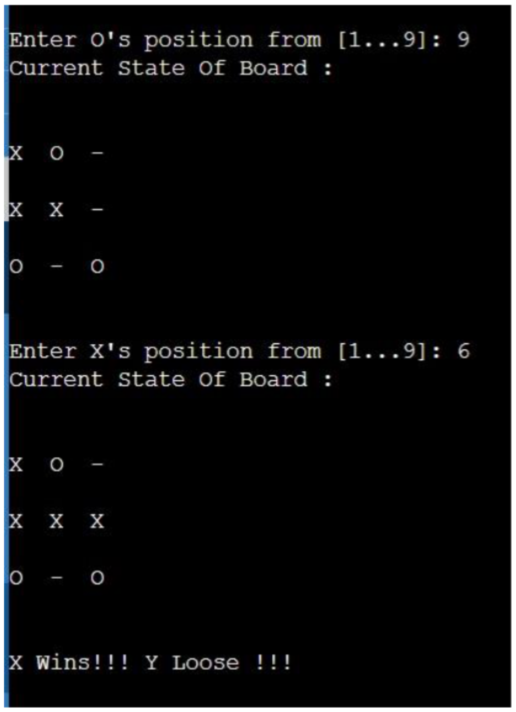
OUTPUT:



```
Enter X's position from [1...9]: 4
Current State Of Board :


X  O  -

X  -  -

-  -  -


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


X  O  -

X  -  -

O  -  -


Enter X's position from [1...9]: 5
Current State Of Board :


X  O  -

X  X  -

O  -  -
```



```
Enter 1 for single player, 2 for multiplayer: 2
Current State Of Board :


-  -  -

-  -  -

-  -  -


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


X  -  -

-  -  -

-  -  -


Enter O's position from [1...9]: 2
Current State Of Board :


X  O  -

-  -  -

-  -  -
```



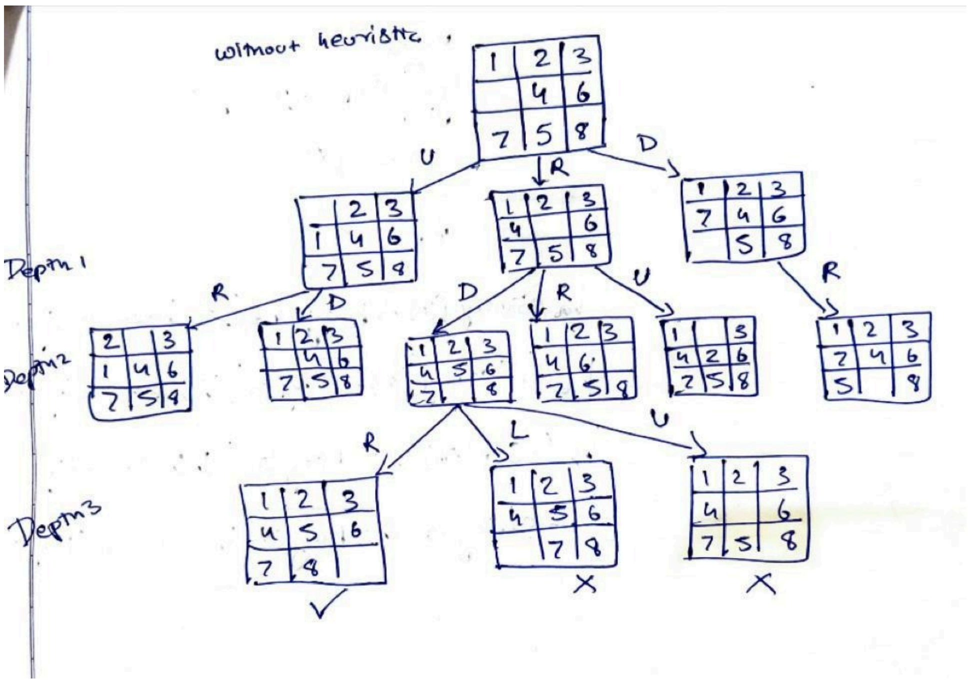
```
Enter O's position from [1...9]: 9
Current State Of Board :


X  O  -

X  X  -

O  -  O


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


X  O  -

X  X  X

O  -  O


X Wins!!! Y Loose !!!
```

Lab-Program-2

Solve 8 puzzle problem.

Objective: The objective of 8-puzzle problem is to reach the end state from the start state by considering all possible movements of the tiles without any heuristic.

Flowchart/State Space Diagram:

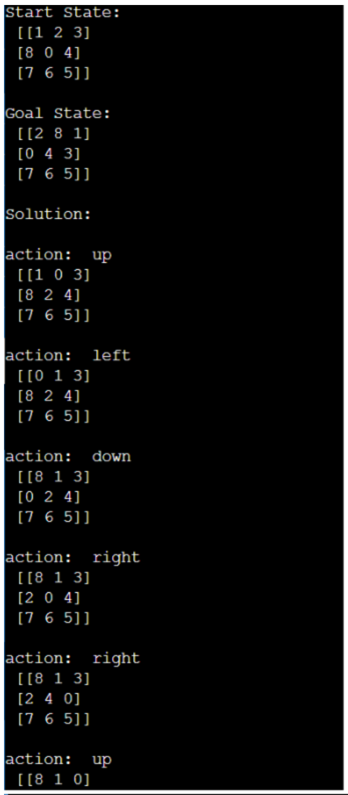


Code:

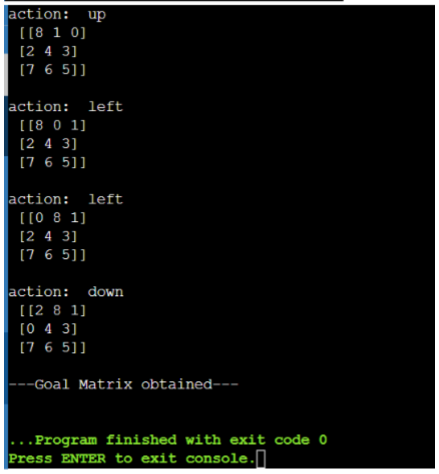
```
import numpy as np
import sys

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


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

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

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

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


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


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

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

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

        return results

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

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

    def solve(self):
        self.num_explored = 0

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

        self.explored = []

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

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

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

            self.explored.append(node.state)

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


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

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

p = Puzzle(start, startIndex, goal, goalIndex)
```

```
p.solve()
p.print()
```

Output:



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

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

Solution:

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

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

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

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

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

action:  up
 [[8 1 0]
```



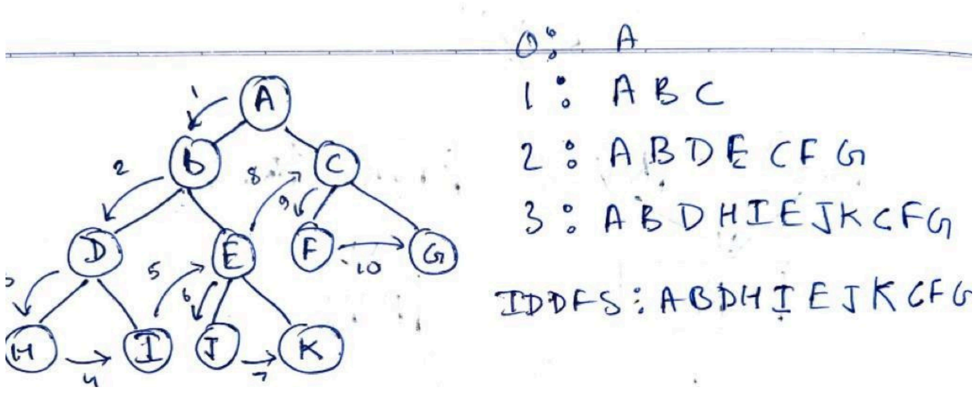
```
action:  up
 [[8 1 0]
 [2 4 3]
 [7 6 5]]

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

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

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

---Goal Matrix obtained---


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab-Program-3

Implement Iterative deepening search algorithm.

Objective: IDDFS combines depth first search's space efficiency and breadth first search's completeness. It improves depth definition, heuristic and score of searching nodes so as to improve efficiency.

Flowchart/State space tree:

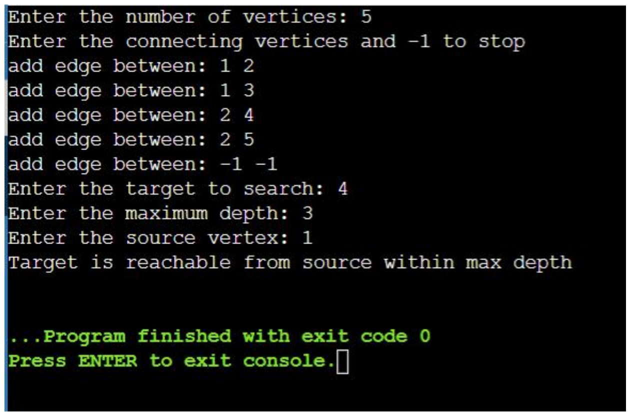


Code:

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

    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DLS(self, src, target, maxDepth):
        if src == target: return True
        if maxDepth <= 0: return False
        for i in self.graph[src]:
            if (self.DLS(i, target, maxDepth -
1)):
                return True
        return False
    def IDDFS(self, src, target, maxDepth):
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False
n = int(input("Enter the number of vertices:
"))
g = Graph(n);
e1 = 1
print("Enter the connecting vertices and -1
to stop")
while e1 != -1:
    e1, e2 = input("add edge between:
").split()
    e1 = int(e1)
    e2 = int(e2)
    if e1 == -1:
        break
    g.addEdge(e1, e2)
target = int(input("Enter the target to
search: "))
maxDepth = int(input("Enter the maximum
depth: "))
  src = int(input("Enter the source vertex: "))
if g.IDDFS(src, target, maxDepth) == True:
    print("Target is reachable from source " +
        "within max depth")
```

```
else:
    print("Target is NOT reachable from
source " +
          "within max depth")
```

Output:



```
Enter the number of vertices: 5
Enter the connecting vertices and -1 to stop
add edge between: 1 2
add edge between: 1 3
add edge between: 2 4
add edge between: 2 5
add edge between: -1 -1
Enter the target to search: 4
Enter the maximum depth: 3
Enter the source vertex: 1
Target is reachable from source within max depth


...Program finished with exit code 0
Press ENTER to exit console.
```

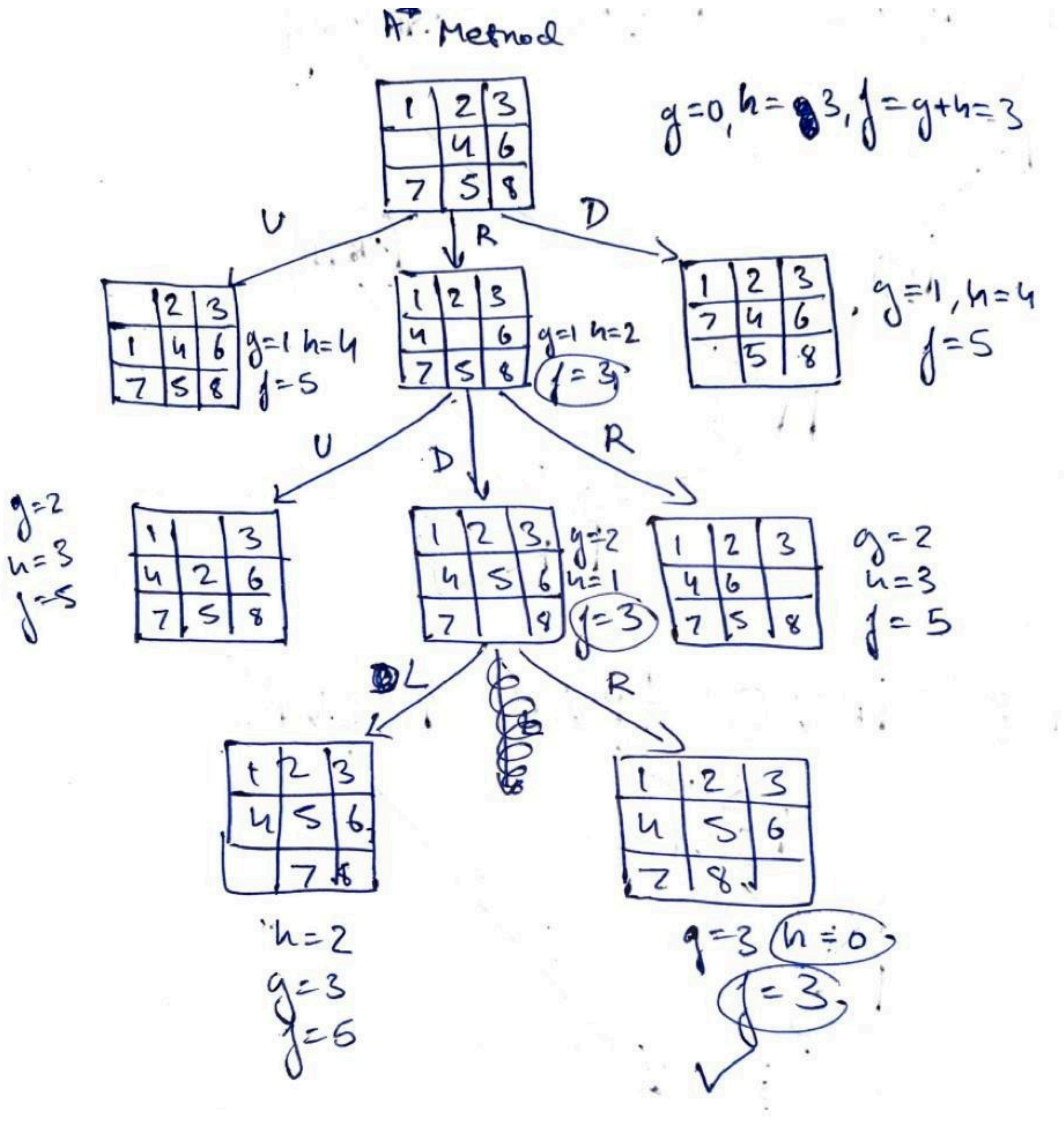
Lab-Program-4

Implement A* search algorithm.

Objective: The a* algorithm takes into account both the cost to go to goal from present state as well the cost already taken to reach the present state.

In 8 puzzle problem, both depth and number of misplaced tiles are considered to take decision about the next state that has to be visited.

Flowchart/State space tree:

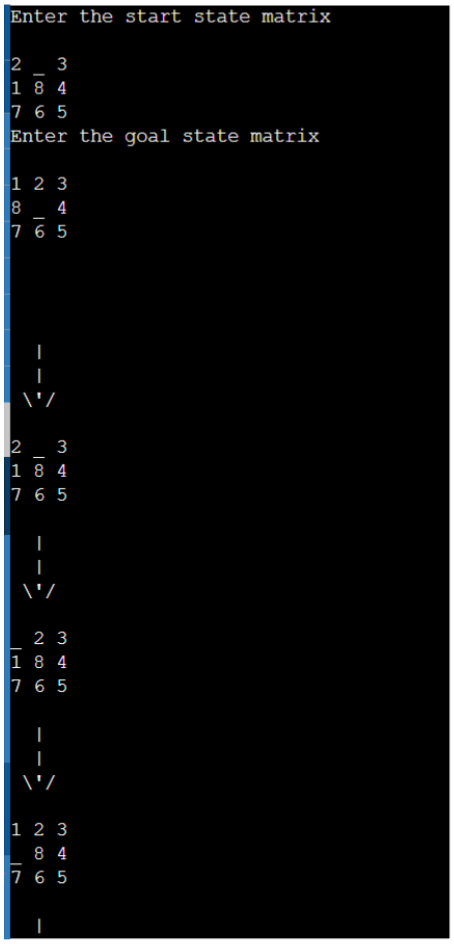


Code:

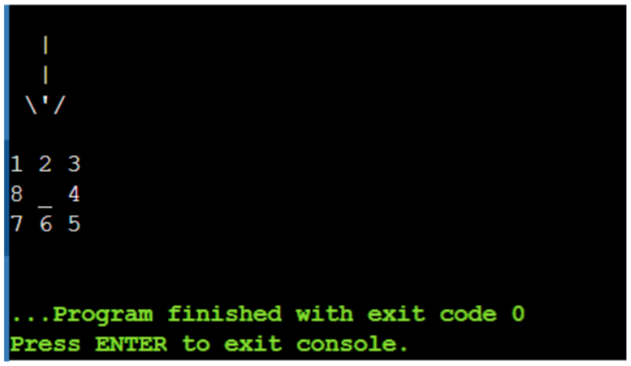
```
class Node:

   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 open list based on f value """

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



puz = Puzzle(3)

puz.process()
```

Output:



```
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

  |
```



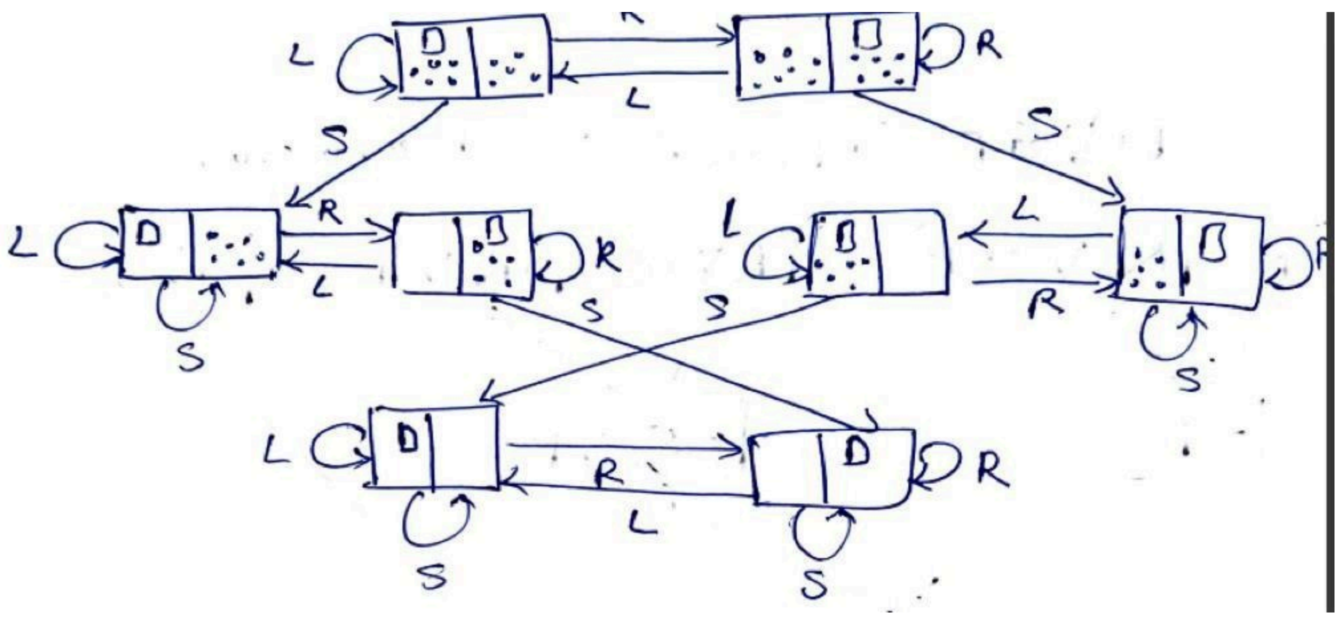
```
  |
  |
 \'/

1 2 3
8 _ 4
7 6 5


...Program finished with exit code 0
Press ENTER to exit console.
```

Program-5

Implement vacuum cleaner agent.

Objective: The objective of the vacuum cleaner agent is to clean the whole of two rooms by performing any of the actions – move right, move left or suck. Vacuum cleaner agent is a goal based agent.

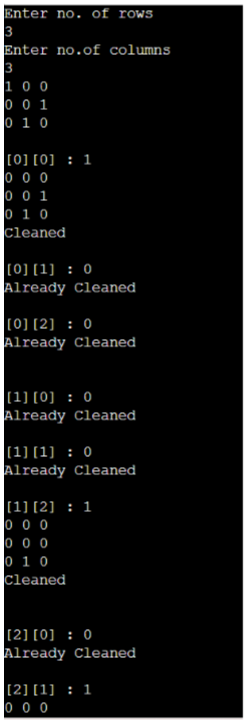
Flowchart/State space tree:



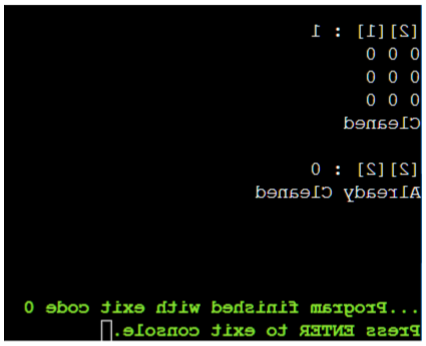
Code:

```
def clean(floor):
    for row in range(len(floor)):
#         print('Floor {} : '.format(row + 1))
        for col in range(len(floor[0])):
            print('[{}][{}] : {}'.format(row, col, floor[row][col]))
            if floor[row][col]:
                floor[row][col] = 0
                print_floor(floor)
                print('Cleaned ')
            else: print('Already Cleaned ')
            print()
        print()

def print_floor(floor): # row, col represent the current vacuum cleaner position
    for i in range(len(floor)):
        for j in range(len(floor[0])):
            print(floor[i][j], end = ' ')
        print()

def main():
    print("Enter no. of rows")
    m = int(input())
    print("Enter no.of columns")
    n = int(input())
    floor = []
    for i in range(0, m):
        a = list(map(int, input().split(" ")))
        floor.append(a)
    print()
    clean(floor)


main()
```

Output:



```
Enter no. of rows
3
Enter no.of columns
3
1 0 0
0 0 1
0 1 0

[0][0] : 1
0 0 0
0 0 1
0 1 0
Cleaned

[0][1] : 0
Already Cleaned

[0][2] : 0
Already Cleaned


[1][0] : 0
Already Cleaned

[1][1] : 0
Already Cleaned

[1][2] : 1
0 0 0
0 0 0
0 1 0
Cleaned


[2][0] : 0
Already Cleaned

[2][1] : 1
0 0 0
```



```
[2][1] : 1
0 0 0
0 0 0
0 0 0
Cleaned

[2][2] : 0
Already Cleaned



...Program finished with exit code 0
Press ENTER to exit console.
```

Lab-Program-6

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

Objective: The objective of this program is to see if the given query entails a knowledge base. A query is said to entail a knowledge base if the query is true for all the models where knowledge base is true.

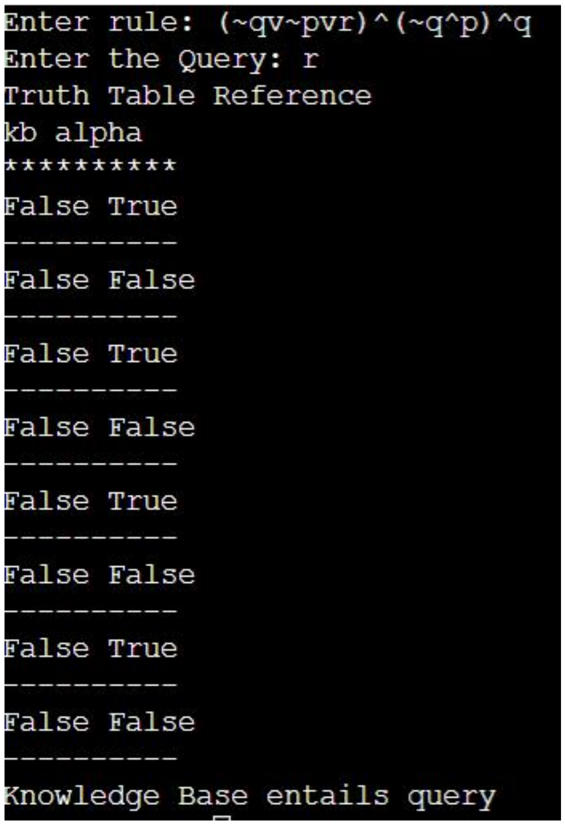
Code:

```
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

#Test 1

input_rules()

ans = entailment()

if ans:

    print("Knowledge Base entails query")

else:

    print("Knowledge Base does not entail query")

#Test 2

input_rules()

ans = entailment()

if ans:

    print("Knowledge Base entails query")

else:

    print("Knowledge Base does not entail query")
```

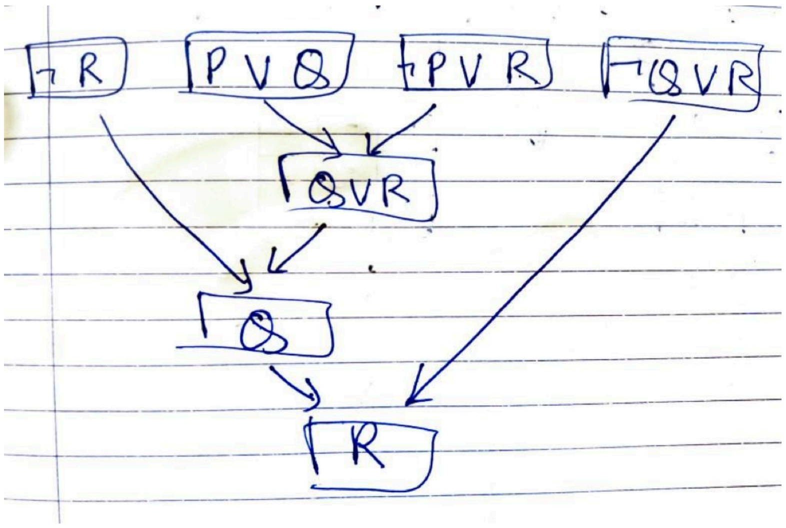
Output:



```
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
----------
Knowledge Base entails query
```

Lab-program-7

Create a knowledgebase using prepositional logic and prove the given query using resolution



Objective: The resolution takes two clauses and produces a new clause which includes all the literals except the two complementary literals if exists. The knowledge base is conjucted with the not of the give query and then resolution is applied.

Code:

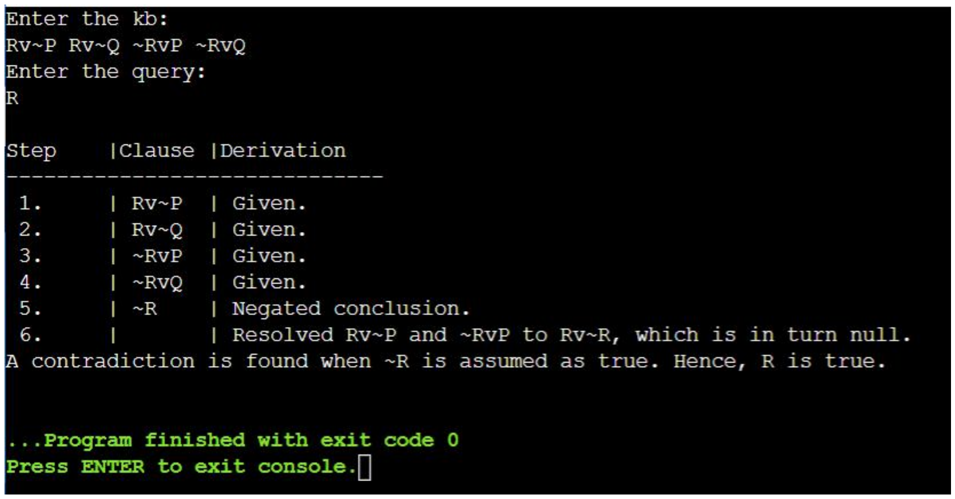
```
import re

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



```
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.


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab-Program-8

Implement unification in first order logic

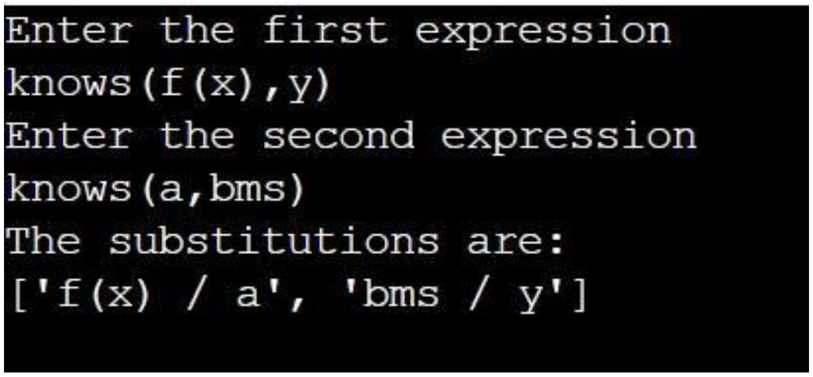
Objective: Unification can find substitutions that make different logical expressions identical. Unify takes two sentences and make a unifier for the two if a unification exist.

Code:

```
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()
```

Output:



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

Lab-Program-9

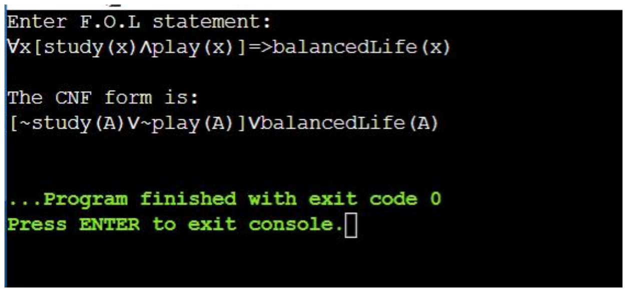
Convert given first order logic statement into Conjunctive Normal Form (CNF).

Objective: FOL logic is converted to CNF makes implementing resolution theorem easier.

Code: import

```
re


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


def getPredicates(string):
    expr = '[a-z~]+\([A-Za-z,]+\)'
return   re.findall(expr,   string)
def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
string = string.replace('~~','')    flag =
'[' in string    string =
string.replace('~[','')    string =
string.strip(']')
    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')

s = list(string) for i, c in enumerate(string): if c ==

'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                                          statement
 )                   for s in statements:           =

 statement.replace(s,
 fol_to_cnf(s))                                    expr =

 '~\[[^]]+
 \]'                 statements = re.findall(expr,

 statement                                          statement
 )                   for s in statements:           =

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()
```

Output:



```
Enter F.O.L statement:
∀x[study(x)∧play(x)]=>balancedLife(x)

The CNF form is:
[~study(A)V~play(A)]VbalancedLife(A)


...Program finished with exit code 0
Press ENTER to exit console.
```

Lab-Program-10

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

Objective: A forward-chaining algorithm will begin with facts that are known. It will proceed to trigger all the inference rules whose premises are satisfied and then add the new data derived from them to the known facts, repeating the process till the goal is achieved or the problem is solved.

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)

                                                  predicate self.params =

                                                  paramsself.predicate =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]:
```

```
            attribute                          =  constants[ke
            s            attributes.replace(key,  y])                expr =

                                                              len(new_lhs)
 f'{predicate}{attributes}'              return  Fact(expr)  if        and

                                                                       def
 all([f.getResult() for f in new_lhs]) else None class KB:       __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: ")

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
for i, f in

kb.tell(t)

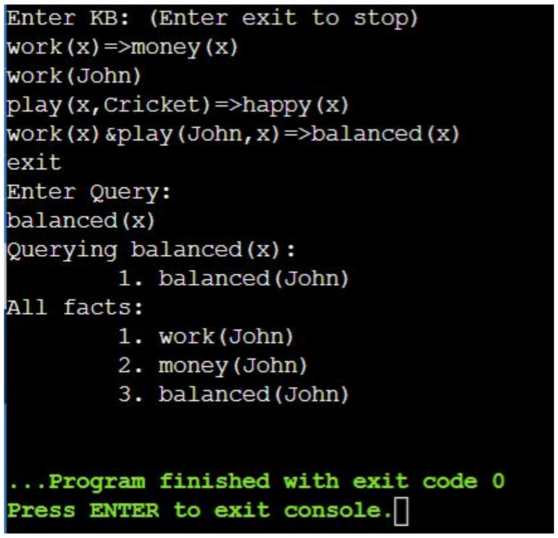
print("Enter Query:")

q = input()

kb.query(q)

kb.display() main()

Output:



```
Enter KB: (Enter exit to stop)
work(x)=>money(x)
work(John)
play(x,Cricket)=>happy(x)
work(x)&play(John,x)=>balanced(x)
exit
Enter Query:
balanced(x)
Querying balanced(x):
        1. balanced(John)
All facts:
        1. work(John)
        2. money(John)
        3. balanced(John)


...Program finished with exit code 0
Press ENTER to exit console.
```