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

Lab Record

Artificial Intelligence

(22CS5PCAIN)

Bachelor of Technology
in
Computer Science and Engineering

Submitted by:

Skanda M Shastry
1BM21CS212

Department of Computer Science and
Engineering B.M.S. College of Engineering
Bull Temple Road, Basavanagudi, Bangalore 560
019 Mar-June 2021

B.M.S. COLLEGE OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Skanda M Shastry(1BM21CS212) during the 5th Semester September-January 2021.

Signature of the Faculty In charge:

Sneha S Bagalkot
Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

Lab-Program-1

(4)

TIC-TAC-TOE

```
import math
import copy
X = 'x'
O = "o"
empty = None
def initial_state():
    return [[empty, empty, empty],
            [empty, empty, empty],
            [empty, empty, empty]]
def player(board):
    count o = 0
    count x = 0
    for y in [0,1,2]:
        for x in board[y]:
            if x == "o":
                count o = count o + 1
            else if x == "x"
                count x = count x + 1
    if count o >= count x:
        return x
    else if count x > count o:
        return o
def action(board):
    free boxes = set()
    for i in [0,1,2]:
        for j in [0,1,2]:
            if board[i][j] = empty:
                free boxes.add((i,j))
    return freeboxes
def result (board . action):
    i = action [0]
    j = action [1]
    if type(action) == list :
```

```
action = (i,j)
if action in actions (board):
    if player (board) == x:
        board [i][j] = x
    elif if player (board) == o:
        board [i][j] = o

return board.

def winner(board):
    if (board[0][0] == board [0][1] == board[0][2] == x or
        board [1][2] == x or board [2][0] == board [2][1] ==
        board [2][2] == x):
        return x
    if (same as above == o):
        return o
    for i in [0, 1, 2]:
        s2 = [ ]
    for j in [0,1,2]:
        s2 . append (board [j][i])
        if (s2 [0] == s2 [1] == s2 [2]):
        return s2 [0]
    strike D= []
    for i in [0,1,2]:
        strike D. append (board [i][i])
        if (strike D[0] == strike D [1] == strike D [2]):
        return strike D [0]
    if board [0][2] == board [1][1] == board [2][0]):
    return board [0][2]
    return None.
def terminal (board):
    full = true
    for i in [0,1,2]:
        for j in board [i]:
        if j in None
        full = false
```

```
if full:
    return True
if (winner(board) in not None):
    return True
    return False

def minimize-helper(board):
    ismaxturn = true if player(board) == X
    else,
        false
  if terminal(board):
    return utility(board)
    scores = []
  for move in actions(board):
    result[board, move])
   scores. Append (minimax helper(board))
   return max(scores) if isMax turn else min(score)

 if is max Turn:
    bestscore = - math.inf
    for move in actions(board):
        result(board. move)
        score = minimax_helper(board)
        board(move[0]].[move[1]] = empty
        if (score > bestscore):
            best score = score
            best move = move
   return best move

 else:
  while not terminal(game-board):
    if player(game-board)==X:
        user-input = input.("Enter"):

    else:
      print -board(game-board)

if winner(gameboard) is not None:
   print("The winner is :{winner(gameboard)}");
   else
 print("It a tie");
```

```
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.e1)).
  for key in constants:
    if constants[key]:
      attributes = attributes.replace(key, constants[key]).
      expr = f'{predicate}{attributes}'

class KB:
   def __init__(self):
      self.facts = set()
      self.implications = set()
   def tell(self, e):
       if '=>' in e:
          self.implications
```

Implement Tic-Tac-Toe Game

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

Code:

```
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:
```

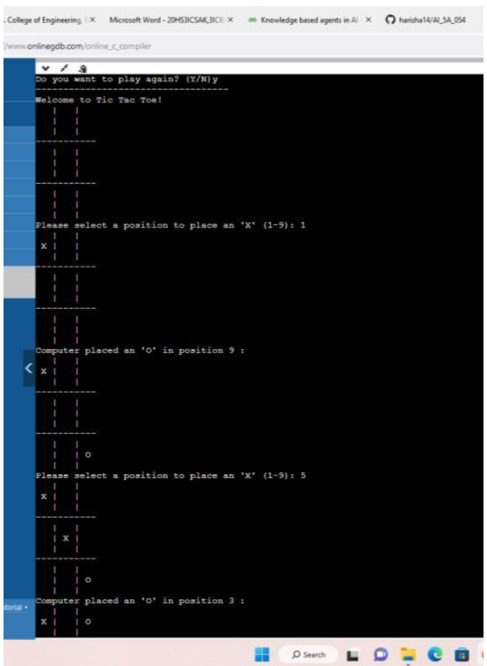
```
            print('Please type a number!')



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:



Lab-Program-2

(3) Puzzle Problem

```
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_state):
    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 ...
```

if [move_it-can not in visited-state].

def gen [state, m, b]
    temp = state.copy()
    if m == 'd'.
    temp [b+3]. temp [b] = temp [b], temp [A+B]
    if m == 'u'
        temp [b-1], temp [b] = temp[b], temp[b],
                                    temp[b+1]

    return temp.

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

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

bfs (src, target).

6/12/23

~~Ex~~
output -
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & & 6 \\ 7 & 5 & 8 \end{bmatrix} \text{state.}$$

goal:
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & \end{bmatrix}$$

→ 1 2 3 4 0 6 7 5 8
→ 1 2 3 4 5 6 7 0 8
→ 1 2 3 4 5 6 7 8 0 .

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.

Code:

```
import numpy as np
import os
class Node:
    def_init_(self, node_no, data, parent, act,
cost):
        self.data = data
        self.parent = parent
        self.act = act
        self.node_no = node_no
        self.cost = cost
def get_initial():
    print("Please enter number from 0-8, no number
should be repeated or be out of this range")
    initial_state = np.zeros(9)
    for i in range(9):
        states = int(input("Enter the " + str(i + 1)
+ " number: "))
        if states < 0 or states > 8:
            print("Please only enter states which are
[0-8], run code again")
            exit(0)
        else:
            initial_state[i] = np.array(states)
    return np.reshape(initial_state, (3, 3))
def find_index(puzzle):
    i, j = np.where(puzzle == 0)
    i = int(i)
    j = int(j)
    return i, j
def move_left(data):
```

```
    i, j = find_index(data)
    if j == 0:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i, j - 1]
        temp_arr[i, j] = temp
        temp_arr[i, j - 1] = 0
        return temp_arr
def move_right(data):
    i, j = find_index(data)
    if j == 2:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i, j + 1]
        temp_arr[i, j] = temp
        temp_arr[i, j + 1] = 0
        return temp_arr
def move_up(data):
    i, j = find_index(data)
    if i == 0:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i - 1, j]
        temp_arr[i, j] = temp
        temp_arr[i - 1, j] = 0
        return temp_arr
def move_down(data):
    i, j = find_index(data)
    if i == 2:
        return None
    else:
        temp_arr = np.copy(data)
        temp = temp_arr[i + 1, j]
        temp_arr[i, j] = temp
        temp_arr[i + 1, j] = 0
        return temp_arr
```

```
def move_tile(action, data):
    if action == 'up':
        return move_up(data)
    if action == 'down':
        return move_down(data)
    if action == 'left':
        return move_left(data)
    if action == 'right':
        return move_right(data)
    else:
        return None
def print_states(list_final):  # To print the final
states on the console
    print("printing final solution")
    for l in list_final:
        print("Move : " + str(l.act) + "\n" + "Result
: " + "\n" + str(l.data) + "\t" + "node number:" +
str(l.node_no))


def write_path(path_formed):  # To write the final
path in the text file
    if os.path.exists("Path_file.txt"):
        os.remove("Path_file.txt")

    f = open("Path_file.txt", "a")
    for node in path_formed:
        if node.parent is not None:
            f.write(str(node.node_no) + "\t" +
str(node.parent.node_no) + "\t" + str(node.cost) +
"\n")
    f.close()
def write_node_explored(explored): # To write all
the nodes explored by the program
    if os.path.exists("Nodes.txt"):
        os.remove("Nodes.txt")

    f = open("Nodes.txt", "a")
    for element in explored:
```

```
        f.write('[')
        for i in range(len(element)):
            for j in range(len(element)):
                f.write(str(element[j][i]) + " ")
        f.write(']')
        f.write("\n")
    f.close()
def write_node_info(visited): # To write all the
info about the nodes explored by the program
    if os.path.exists("Node_info.txt"):
        os.remove("Node_info.txt")

    f = open("Node_info.txt", "a")
    for n in visited:
        if n.parent is not None:
            f.write(str(n.node_no) + "\t" +
str(n.parent.node_no) + "\t" + str(n.cost) + "\n")
    f.close()
def path(node):  # To find the path from the goal
node to the starting node
    p = [] # Empty list
    p.append(node)
    parent_node = node.parent
    while parent_node is not None:
        p.append(parent_node)
        parent_node = parent_node.parent
    return list(reversed(p))
def path(node): # To find the path from the goal
node to the starting node
    p = [] # Empty list
    p.append(node)
    parent_node = node.parent
    while parent_node is not None:
        p.append(parent_node)
        parent_node = parent_node.parent
    return list(reversed(p))
def path(node):  # To find the path from the goal
node to the starting node
    p = []  # Empty list
```

```
    p.append(node)
    parent_node = node.parent
    while parent_node is not None:
        p.append(parent_node)
        parent_node = parent_node.parent
    return list(reversed(p))
def check_correct_input(l):
    array = np.reshape(l, 9)
    for i in range(9):
        counter_appear = 0
        f = array[i]
        for j in range(9):
            if f == array[j]:
                counter_appear += 1
        if counter_appear >= 2:
            print("invalid input, same number entered
2 times")
            exit(0)
def check_solvable(g):
    arr = np.reshape(g, 9)
    counter_states  =  0
    for i in range(9):
        if not arr[i] == 0:
            check_elem = arr[i]
            for x in range(i + 1, 9):
                if check_elem < arr[x] or arr[x] ==
0:
                    continue
                else:
                    counter_states += 1
    if counter_states % 2 == 0:
        print("The puzzle is solvable, generating
path")
    else:
        print("The puzzle is insolvable, still
creating nodes")
k = get_initial()

check_correct_input(k)
```

```
check_solvable(k)

root = Node(0, k, None, None, 0)

# BFS implementation call
goal, s, v = exploring_nodes(root)

if goal is None and s is None and v is None:
    print("Goal State could not be reached, Sorry")
else:
    # Print and write the final output
    print_states(path(goal))
    write_path(path(goal))
    write_node_explored(s)
    write_node_info(v)
```

Output:

```
Please enter number from 0-8, no number should be repeated or be out of this range
Enter the 1 number: 1
Enter the 2 number: 3
Enter the 3 number: 2
Enter the 4 number: 5
Enter the 5 number: 4
Enter the 6 number: 6
Enter the 7 number: 0
Enter the 8 number: 7
Enter the 9 number: 8
The puzzle is solvable, generating path
Exploring Nodes
Goal_reached
printing final solution
Move : None
Result :
[[1. 3. 2.]
 [5. 4. 6.]
 [0. 7. 8.]]    node number:0
Move : up
Result :
[[1. 3. 2.]
 [0. 4. 6.]
 [5. 7. 8.]]    node number:1
Move : right
Result :
[[1. 3. 2.]
 [4. 0. 6.]
 [5. 7. 8.]]    node number:5
```

Lab-Program-3

→ Analyse Iterative deepening -

```
def iterative - deep - search (src, target):
    depth - Limit = 0
    while True :
        result = depth - limited - search (src, target, depth
        If result is not None :
            print ("success")
            return
        depth - limit += 1
        if depth - limit > 30:
            print ("solution Not found within depth Limit");
        return
def depth - limited - search (src, target, depth, visited):
    if src == target :
        print state (src)
        return src
    if depth - Limit = 0:
        Return None
    visited - states . append (src)
    Poss - moves - to - do = possible - moves( src, vis)
    for move in Poss - moves - to - do:
        if move not in visited:
            Print - State (move)

def get - neighbours (state):
    neighbours = []
    empty - index = state . index (0)
    row, col = divmod (empty - index ,3)
for move in [(0,1) , (1,0) ,(0,-1) (-1,0)]:
    new - row, new - col = new row + move [0], col + move [1]
if 0 <= new - row
```

```
return None
if src = long ;
    return Path +(src)
for action in actions(src):
    new-state = apply-action(src, action)
    result = depth. limited dfs (new-src, basel, depth-limit,
                                    path-1, [src])
    if result:
        return result.
return false.
def iddfs (src, target, max-depth):
    for depth - limit in range (max-depth-1):
    result = depth - limit - dfs (src, target, depth, limit)
    if result :
        return result.
    return false.
```

output :

src i = [1,2,3,4,5,6,7,8]
length = [1,2,3,4,5,-1,6,7,8]
depth = 1
false
src2 = [3,5,2,8,7,6,9,1,-1]
target2 = [-1,3,7,8,1,5,4,6,2]
depth2 = 1
false
src3 = [1,1,3,-1,4,5,6,7,8]
target3 = [1,2,3,6,4,5,-1,7,8]
depth3 = 1
True.

2 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.

Code:

```
import copy
inp=[[1,2,3],[4,-1,5],[6,7,8]]
out=[[1,2,3],[6,4,5],[-1,7,8]]
def move(temp, movement):
  if movement=="up":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
        if i!=0:
            temp[i][j]=temp[i-1][j]
            temp[i-1][j]=-1
          return temp
  if movement=="down":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
        if i!=2:
            temp[i][j]=temp[i+1][j]
            temp[i+1][j]=-1
          return temp
  if movement=="left":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
        if j!=0:
            temp[i][j]=temp[i][j-1]
            temp[i][j-1]=-1
          return temp
```

```
  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=2:
            temp[i][j]=temp[i][j+1]
            temp[i][j+1]=-1
          return temp
def ids():
  global  inp
  global  out
  global flag
  for limit in range(100):
    print('LIMIT -> '+str(limit))
    stack=[]
    inpx=[inp,"none"]
    stack.append(inpx)
    level=0
    while(True):
      if len(stack)==0:
        break
      puzzle=stack.pop(0)
      if level<=limit:
        print(str(puzzle[1])+" --> "+str(puzzle[0]))
        if(puzzle[0]==out):
          print("Found")
          print('Path cost='+str(level))
          flag=True
          return
        else:
          level=level+1
          if(puzzle[1]!="down"):
          temp=copy.deepcopy(puzzle[0])
          up=move(temp, "up")
          if(up!=puzzle[0]):
              upx=[up,"up"]
              stack.insert(0, upx)
```

```
          if(puzzle[1]!="right"):
            temp=copy.deepcopy(puzzle[0])
            left=move(temp, "left")
            if(left!=puzzle[0]):
            leftx=[left,"left"]
              stack.insert(0, leftx)
          if(puzzle[1]!="up"):
          temp=copy.deepcopy(puzzle[0])
          down=move(temp, "down")
          if(down!=puzzle[0]):
              downx=[down,"down"]
              stack.insert(0, downx)
          if(puzzle[1]!="left"):
            temp=copy.deepcopy(puzzle[0])
         right=move(temp, "right")
            if(right!=puzzle[0]):
            rightx=[right,"right"]
            stack.insert(0, rightx)
print('~~~~~~~~~~~~~ IDS ~~~~~~~~~~~~~')
ids()
```

```
import copy
inp=[[1,2,3],[4,-1,5],[6,7,8]]
out=[[1,2,3],[6,4,5],[-1,7,8]]
def move(temp, movement):
  if movement=="up":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
        if i!=0:
            temp[i][j]=temp[i-1][j]
            temp[i-1][j]=-1
          return temp
  if movement=="down":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
        if i!=2:
            temp[i][j]=temp[i+1][j]
```

```
          temp[i+1][j]=-1
        return temp
  if movement=="left":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=0:
            temp[i][j]=temp[i][j-1]
            temp[i][j-1]=-1
          return temp
  if movement=="right":
    for i in range(3):
      for j in range(3):
        if(temp[i][j]==-1):
          if j!=2:
            temp[i][j]=temp[i][j+1]
            temp[i][j+1]=-1
          return temp
def ids():
  global  inp
  global  out
  global flag
  for limit in range(100):
    print('LIMIT -> '+str(limit))
    stack=[]
    inpx=[inp,"none"]
    stack.append(inpx)
    level=0
    while(True):
      if len(stack)==0:
        break
      puzzle=stack.pop(0)
      if level<=limit:
        print(str(puzzle[1])+" --> "+str(puzzle[0]))
        if(puzzle[0]==out):
          print("Found")
          print('Path cost='+str(level))
          flag=True
          return
        else:
          level=level+1
          if(puzzle[1]!="down"):
          temp=copy.deepcopy(puzzle[0])
```

```
            up=move(temp, "up")
            if(up!=puzzle[0]):
              upx=[up,"up"]
              stack.insert(0, upx)
          if(puzzle[1]!="right"):
            temp=copy.deepcopy(puzzle[0])
            left=move(temp, "left")
            if(left!=puzzle[0]):
            leftx=[left,"left"]
              stack.insert(0, leftx)
          if(puzzle[1]!="up"):
          temp=copy.deepcopy(puzzle[0])
          down=move(temp, "down")
          if(down!=puzzle[0]):
              downx=[down,"down"]
              stack.insert(0, downx)
          if(puzzle[1]!="left"):
            temp=copy.deepcopy(puzzle[0])
         right=move(temp, "right")
            if(right!=puzzle[0]):
            rightx=[right,"right"]
            stack.insert(0, rightx)
print('~~~~~~~~~~~~~ IDS ~~~~~~~~~~~~~')
ids()
```

Output:

```
#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)
```

```
False
```

```
#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)
```

```
False
```

```
# 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)
```

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

for i in range(1, 100):
    val = iddfs(src,target,i)
    print(i, val)
    if val == True:
        break
```

```
1 False
2 False
3 False
4 False
5 False
6 False
7 False
8 False
9 False
10 False
11 False
12 False
13 False
14 False
15 False
16 False
17 False
18 False
19 False
20 False
21 False
22 False
23 False
24 False
```

Lab Program 4

A* Algorithm

```
import heapq
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval
    def gen_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).
        y2 < len(self.data) and y2 >= 0 and y2 < len(self.data)
            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:
    start = Node(start_data, 0, 0)
    start.fval = self.f(start, goal_data)
    self.open.append(start)
    print("\n\n")
    while(True):
        cur = self.open[0]
        print("")
        print(" | ")
        print(" \\\'/ \n")
        for i in cur.data
            for j in i:
                print(j, end=" ")
            print("")
        if self.h(cur.data, goal_data) == 0:
            break.
        for i in cur.generate_child():
            i.fval = self.f(i, goal_data)
            self.open.append(i)
        self.closed.append(cur)
        del self.open[0]
        self.open.sort(key = lambda x: x.fval, reverse = False)
```

start_state = [['1','2','3'], ['-','4','6'], ['7','5','8']]
goal_state = [['1','2','3'], ['4','5','6'], ['7','8','-']]
Puz = Puzzle (3)
Puz.process (start_state, goal_state)

O/P -
1 2 3
- 4 6
7 5 8

output -
1 2 3
4 5 6
7 8 -

20/12/23

Propositional logic

```
def evaluate_expression(q,p,r):
    expression_result = ((not q or not p or r) and (not q and p) and q)
    return expression_result

def generate_truth_table():
    Print ("q|p|r| Expression(KB) | query (r)")
    Print ("---|---|---|-----|----")

    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(q,p,r)
                query_result = r
                print (f"{q}|{p}|{r}| {expression_result}
                        |{query_result}")

def query_entails_knowledge ():
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression (q,p,r)
                query_result = r
                if expression_result and not query_result:
                    return False
    return True

def main ():
    generate_truth_table()
    if query_entails_knowledge ():
```

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.

Code:

```
def print_b(src):
    state = src.copy()
    state[state.index(-1)] = ' '
    print(
f"""
{state[0]} {state[1]} {state[2]}
{state[3]} {state[4]} {state[5]}
{state[6]} {state[7]} {state[8]}
 “””
    )
def h(state, target):
    count = 0
    i = 0
    for j in state:
        if state[i] != target[i]:
            count = count+1
    return count
def astar(state, target):
    states = [src]
    g = 0
    visited_states = []
    while len(states):
        print(f"Level: {g}")
        moves = []
        for state in states:
            visited_states.append(state)
            print_b(state)
            if state == target:
```

```
                print("Success")
                return
            moves += [move for move in possible_moves(
                state, visited_states) if move not in moves]
        costs = [g + h(move, target) for move in moves]
        states = [moves[i]
                  for i in range(len(moves)) if costs[i] == min(costs)]
        g += 1
    print("Fail")
def possible_moves(state,  visited_state):
    b = state.index(-1)
    d = []
    if b - 3 in range(9):
        d.append('u')
    if b not in [0, 3, 6]:
        d.append('l')
    if b not in [2, 5, 8]:
        d.append('r')
    if b + 3 in range(9):
        d.append('d')
    pos_moves = []
    for m in d:
        pos_moves.append(gen(state, m, b))
    return [move for move in pos_moves if move not in visited_state]
def gen(state, m, b):
    temp = state.copy()
    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]
    if m == 'd':
        temp[b + 3], temp[b] = temp[b], temp[b + 3]
    return temp
src = [1, 2, 3, -1, 4, 5, 6, 7, 8]
target = [1, 2, 3, 4, 5,6, 7, 8,-1]
```

astar(src, target)

Output:

```
Enter the start state matrix

1 0 1 0
1 0 0 1
1 1 1 1
Enter the goal state matrix

1 1 0 1
1 0 0 1
1 1 1 0
|
  |
 \'/

1 0 1 0
1 0 0 1
1 1 1 1
```

Lab Program 5

(2)

Vaccum cleaner agent

```
def vaccum - world():
    goal -state = {'A': '0' : 'B' : '0' },
    cost = 0
    location-input = input (" Enter location of vaccum")
    status.input = input ("Enter status of other room")
if location -input == 'A' :
    Print ("Vaccum is placed in location A")
    if status - input == '1' :
        print ("Location A is Dirty : ")
        goal = state ['A'] = '0'
        cost += 1
    print ("cost for cleaning A" + str (cost))
    print ("location A has been cleaned")

else :
    Print ("No action" + str(cost))
    print ("LOC B has been cleaned")

else :
    print (" no action "):
if status - input == '0' :
    print ("LOC A is already clean")
    if status _ input = complement == '1' :
        Print ("LOC B is dirty")
        Print ("moving Right to the LOC B")
        cost += 1
        print ("cost for suck " + str(cost))
        print ("LOC A has been cleaned")
else :
    Print(cost)
    print("LOC B is already clean: ")
```

3

```
else :
  print (cost)
  print ("LocB is already clean :")
  if state - input : complement == '1' :
    print ("Loc A is Dirty")
    print ("moving left to . the loc A")
    cost += 1
    print ("cost for suck " + str (cost))
    print ("LocA has been cleaned.")
  else :
    print ("No action" + str (cost))
    print ("LocA is already clean")
  print ("goal . state : ")
  print (goal - state)
  print ("performing measurement" + str (cost))
vaccum - world ()
```

O/P-
```
Enter location : A
Enter state of A 1
Enter status of other room : 1
Initial location of A is 1
vaccum in A
A is dirty
cost for cleaning is 1
A is cleaned
B is dirty
move right to B
cost is suck 3
GOAL STATE :
{'A': '0', 'B': '0'}
Performance measurement : 3
```

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.

Code:

```
def vacuum_world():

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

    location_input = input("Enter Location of Vacuum: ")
    status_input = input("Enter status of " + location_input+ " : ")
    status_input_complement = input("Enter status of other room : ")
    print("Initial Location Condition {A : " + str(status_input_complement) + ",
B : " + str(status_input) + " }" )

    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            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':
                print("Location B is Dirty.")
                print("Moving right to the Location B. ")
                cost += 1
                print("COST for moving RIGHT " + str(cost))
                goal_state['B'] = '0'
                cost += 1
                print("COST for SUCK " + str(cost))
                print("Location B has been Cleaned. ")
```

```
        else:
            print("No action" + str(cost))
            print("Location B is already clean.")

    if status_input == '0':
        print("Location A is already clean ")
        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving RIGHT to the Location B. ")
            cost += 1
            print("COST for moving RIGHT " + str(cost))
            goal_state['B'] = '0'
            cost += 1
            print("Cost for SUCK" + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action " + str(cost))
            print(cost)
            print("Location B is already clean.")

else:
    print("Vacuum is placed in location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'
        cost += 1
        print("COST for CLEANING " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to the Location A. ")
            cost += 1
            print("COST for moving LEFT " + str(cost))
            goal_state['A'] = '0'
            cost += 1
            print("COST for SUCK " + str(cost))
```

```
            print("Location A has been Cleaned.")

        else:
            print(cost)
            print("Location B is already clean.")

            if status_input_complement == '1':
                print("Location A is Dirty.")
                print("Moving LEFT to the Location A. ")
                cost += 1
                print("COST for moving LEFT " + str(cost))
                goal_state['A'] = '0'
                cost += 1
                print("Cost for SUCK " + str(cost))
                print("Location A has been Cleaned. ")
            else:
                print("No action " + str(cost))
                print("Location A is already clean.")

    print("GOAL STATE: ")
    print(goal_state)
    print("Performance Measurement: " + str(cost))

vacuum_world()
```

Output:

```
Enter Location of Vacuum: A
Enter status of A : 0
Enter status of other room : 1
Initial Location Condition {A : 1, B : 0 }
Vacuum is placed in Location A
Location A is already clean
Location B is Dirty.
Moving RIGHT to the Location B.
COST for moving RIGHT 1
Cost for SUCK2
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 2
```

Lab Program 6

```
start_state = [['1','2','3'], ['-','4','6'], ['7','5','8']]
goal_state = [['1','2','3'], ['4','5','6'], ['7','8','-']]
puz = Puzzle(3)
puz.process(start_state, goal_state)
```

o/p -

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

output -

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

20/12/23

Propositional logic

```
def evaluate_expression(q,p,r):
    expression_result = ((not q or not p or r) and (not q and p) and q)
    return expression_result

def generate_truth_table():
    print("q|p|r| Expression(KB) |query(r)")
    print("---|---|---|-----|---")

    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(q,p,r)
                query_result = r
                print(f"{q}|{p}|{r}| {expression_result}
                |{query_result}")

def query_entails_knowledge():
    for q in [True, False]:
        for p in [True, False]:
            for r in [True, False]:
                expression_result = evaluate_expression(q,p,r)
                query_result = r
                if expression_result and not query_result:
                    return False
    return True

def main():
    generate_truth_table()
    if query_entails_knowledge():
```

output:

① Enter rule = (~q v ~p v r) ∧ (~q ∧ p) ∧ q
query = r

False True
False False
False True
False False
False True
False False
False False
False False

Query satisfies the knowledge.

② enter rule: (p ∨ q) ∧ (~r v p)
query : p∧r

1. . . truth table . . .]

True True
True False

The knowledge base does not entail query.

Propositional-2

```
import re
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nstep\t|clause\t|derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t|{step}\t|{steps[step]}\t')
        i += 1

def negate(term):
    return f'{term}' if 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.

    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]}',

    return steps
rules = 'Rv~P Rv~Q ~RvP ~RvQ'
goal = 'R'
```

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

output:

(1) Enter rule = (~q v ~p v r) ^ (~q ^ p) ^ q

query = r

False True
False False
False True
False False
False True
False False
False False
False False

Query entails the knowledge.

(2) enter rule: (p v q) ^ (~r v p)

query : p ^ r

[... truth table ...]

True True
True False

The knowledge base does not entail query.

Propositional-2

```
import re
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nstep\t|clause\t|derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t|{step}\t|{steps[step]}\t')
        i += 1
def negate(term):
    return f'~{term}' if negate(i) in terms2:
        t1 = [t for t in terms1 if t != c]
        t2 = [t for t in terms2 if t != negate(c)]
        gen = t1 + t2
    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]}'
    return steps
rules = 'Rv~P Rv~Q ~RvP ~RvQ'
goal = 'R'
```

goal = 'R'
main (Rules, goal)
rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q'
main (rules, 'R')

Output -
Enter 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

A contradiction is found when ~R is assumed as true. Hence ~R is true.

27/12/23

Unification

```
def unify(expr1, expr2):
    func1, args1 = expr1.split('(', 1)
    func2, args2 = expr2.split('(', 1)

    if func1 != func2:
        print("Expressions cannot be unified. Different")
        return None

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

substitution = {}

for a1, a2 in zip(args1, args2):
    if a1.islower() and a2.islower() and a1 != a2:
        substitution[a1] = a2
    elif a1.islower() and not a2.islower() and a1 != a2:
        substitution[a1] = a2
    elif not a1.islower() and a2.islower():
        substitution[a2] = a1
    elif a1 != a2:
        return None
return substitution.

def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
```

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 conjuncted with the not of the give query and then resolution is applied.

Code:

```
def disjunctify(clauses):
    disjuncts = []
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
    return disjuncts

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

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

def checkResolution(clauses, query):
    clauses += [query if query.startswith('~') else '~' + query]
    proposition = '^'.join(['(' + clause + ')' for clause in clauses])
    print(f'Trying to prove {proposition} by contradiction ... ')

    clauses = disjunctify(clauses)
    resolved = False
```

```
    new = set()

    while not resolved:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]
        for (ci, cj) in pairs:
            resolvant = resolve(ci, cj)
            if not resolvant:
                resolved = True
                break
            new = new.union(set(resolvents))
        if new.issubset(set(clauses)):
            break
        for clause in new:
            if clause not in clauses:
                clauses.append(clause)

    if resolved:
        print('Knowledge Base entails the query, proved by resolution')
    else:
        print("Knowledge Base doesn't entail the query, no empty set produced
after resolution")
clauses = input('Enter the clauses ').split()
query = input('Enter the query: ')
checkResolution(clauses, query)
```

Output:

```
#Test1
TELL(['implies', 'p', 'q'])
TELL(['implies', 'r', 's'])
ASK(['implies',['or','p','r'], ['or', 'q', 's']])
```

```
True
```

```
CLEAR()
```

```
#Test2
TELL('p')
TELL(['implies',['and','p','q'],'r'])
TELL(['implies',['or','s','t'],'q'])
TELL('t')
ASK('r')
```

```
True
```

```
CLEAR()
```

```
#Test3
TELL('a')
TELL('b')
TELL('c')
TELL('d')
ASK(['or', 'a', 'b', 'c', 'd'])
```

Lab Program 8

```
goal = 'R'
main (Rules, goal)
rules = 'PVQ PVR ~PVR RVS RV~Q ~SV~Q'
main (rules, 'R')
```

Output-

Enter 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

A contradiction is found when ~R is assumed as true.
Hence ~R is true.

27/12/23

Unification

```
def unify(expr1, expr2):
    func1, args1 = expr1.split('(',1)
    func2, args2 = expr2.split('(',1)

    if func1 != func2:
        print("Expressions cannot be unified. Different")
        return None

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

    substitution = {}

    for a1, a2 in zip(args1, args2):
        if a1.islower() and a2.islower() and a1 != a2:
            substitution[a1] = a2
        elif a1.islower() and a2.islower() and a1 != a2:
            substitution[a1] = a2
        elif not a1.islower() and a2.islower():
            substitution[a2] = a1
        elif a1 != a2:
            return None
    return substitution.

def apply_substitution(expr, substitution):
    for key, value in substitution.items():
        expr = expr.replace(key, value)
```

```
if _name_ == "_main_":
    expr1 = input("Enter the first expression: ")
    expr2 = input("Enter the second expression: ")

    subs = unify(expr1, expr2)

    if substitution:
        print("The substitutions are:")
        for key, value in substitution.items():
            print(f'{key} {value}')

        expr1_result = apply_substitution(expr1, substitution)
        expr2_result = apply_substitution(expr2, substitution)

        print(f'unified expression1: {expr1_result}')
        print(f'unified expression 2: {expr2_result}')
```

o/p - Enter the first expression : Sin(x)
Enter the second expression : Sin(a)
Expressions cannot be unified. Different function

10/1/24

Enter first expression : f(x,y)
enter second expression : f(a,b)
Expression cannot be unified. different function

→ FOL to CNF

17/01/24

```
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 skolemization(statement):
    skolem_constants = [f'{chr(c)}' for c in range(ord('A'),
    ord('Z')+1)]
    matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],
                skolem_constants.pop(0))
    return statement
```

```
import re

def fol_to_cnf(fol):
    statement = fol.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
```

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

if __name__ == "__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:

```
Enter the first expression
king(x)
Enter the second expression
king(john)
The substitutions are:
['john / x']
```

Lab Program 9

```
if __name__ == "__main__":
    expr1 = input("Enter the first expression: ")
    expr2 = input("Enter the second expression: ")

    subs = unify(expr1, expr2)

    if substitution:
        print("The substitutions are:")
        for key, value in substitution.items():
            print(f'{key} {value}')

        expr1_result = apply_substitution(expr1, substitution)
        expr2_result = apply_substitution(expr2, substitution)
        print(f'unified expression1: {expr1_result}')
        print(f'unified expression 2: {expr2_result}')
```

o/p - Enter the first expression : sin(x)
Enter the second expression : sin(a)
Expressions cannot be unified. Different functions

10/1/24

Enter first expression : f(x, y)
Enter second expression : g(a, b)
Expression cannot be unified. different functions

→ FOL to CNF

17/01/24

```
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 Skolemization(statement):
    skolem_constants = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    matches = re.findall('[∃].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        for predicate in getPredicates(statement):
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1], skolem_constants.pop(0))
    return statement

import re

def fol_to_cnf(fol):
    statement = fol.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
```

```
statement = statement.replace( ... (s)) ,
while '~' in statement:
        i = statement.index('~')
        br = statement.index('[') if '[' in statement

else 0
new-statement = '~' + statement[br:i] + ')' + statement[i+1:]
statement = statement[:br] + new-statement

if br>0 else
new-statement

return skolemization(statement)

Print (fol_to_cnf ("bird(x) => ~fly(x)"))
Print (fol_to_cnf ("∃x (bird(x) => ~fly(x))"))
```

O/P ~ ~bird (x) | ~fly(x)

[~bird (A) | ~fly (A)]

17/1/24

→ Query - FOL

```
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
        self.predicate = predicate
        self.params = params
        self.result = any( self.get constants())
    def splitExpr (self, expression):
        predicate = getPredicates ( expression ) [0]
        params = getAttributes

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

class Implication:
```

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('[A∃].', 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 '~A' in statement:
        i = statement.index('~A')
        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] = 'A', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[A','[~A')
    statement = statement.replace('~[∃','[~∃')
    expr = '(~[AV∃].)'
    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()

Output:

```
main()
```

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

```
main()
```

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

Lab Program 10

```
Statement = Statement . replace ( ... (s)),
while '~' in Statement:
        i = Statement . index('~')
        br = Statement . index ('[') if '[' in Statement
else 0
new-Statement = '~' + Statement [br:i] + ']' + Statement[i+1:]
Statement = Statement [:br] + new-Statement
if br>0 else
new-Statement
return Skolemization [Statement]
Print (fol-to-cnf ("bird(x) => ~fly(x)"))
Print (fol-to-cnf ("∃x [bird(x) => ~fly(x)]"))
```

O/P - ~bird (x) | ~fly(x)

[~bird (A) | ~fly (A)]

17/1/24

→ Query - FOL

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

def get Attribute (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
  self.predicate = Predicate
  self.Params = Params
  self.result = any( self.get Constants())
  def Split Expr (self, expression):
    Predicate = getPredicates ( expression ) [0]
    Params = get Attributes

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

class Implication:
```

```
res = :- evalute (sist. fact)

    if res:
        self.facts . add (res).

Kb_ = KB()
Kb_.tell ('king(x) & greedy (x) => evil(x)')
Kb_tell ('king(john)')
Kb_tell ('king (Richard)')
Kb_query ('evil (x)')
```

O/P - John is evil.

31/7/24

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)
        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 ask(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 the number of FOL expressions present in KB:")
n = int(input())
print("Enter the expressions:")
for i in range(n):
    fact = input()
    kb.tell(fact)
print("Enter the query:")
query = input()
kb.ask(query)
kb.display()
```

Output:

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