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

Lab Record
Artificial Intelligence
(22CS5PCAIN)

Submitted in partial fulfillment for the 6th Semester Laboratory

Bachelor of Technology

in

Computer Science and Engineering

Submitted by:

Name: Shivani Sathyanarayanan
USN: 1BM21CS203

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 (20CS5PCAIP) laboratory has been carried out by Shivani Sathyanarayanan (1BM21CS203) during the 5th Semester September-January 2021.

Signature of the Faculty Incharge:

Name: Dr. Pallavi G B
Designation: Assistant Professor
Department of Computer Science and Engineering
B.M.S. College of Engineering, Bangalore

Table of Contents

1. Tic Tac Toe

LAB - 1 TICTAC TOE

```
import random

board = ['' for _in range (10)]

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

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

def printBoard (board):
  #print ('|1')
  print(' '+ board[1] + '|'+ board[2] + '|' + board [3])
  #print ('11')
  print ('-----')
  #print ('11')
  #print (' '+ board [4] + '|' + board [5] + '|' + board [6])
  #print ('11')
  print('-----')
  #print ('11')
  print (' ' + board[7] + '|' + board[8] + '|' + board [9])
  #print ('11')

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():
  global board
  run = True
  while run:
    move = int(move)
    if 1 <= move <= 9:
      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 ValueError:
    print('Please type a Number!')

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

  for let in ['O', 'X']:
    for i in possibleMoves:
      boardCopy = Board[:]
      boardCopy[i] = let
      if isWinner(boardCopy, let):
```

store
67

```
    return ;

corners0pen = [i for i in possibleMoves if i in [1,3,7,9]]
if cornersOpen:
    return selectRandom(cornersOpen)
if 5 in possibleMoves:
    return 5

edgesOpen = [i for i in possibleMoves if i in [2,4,6,8]]
if edgesOpen:
    return selectRandom(edgesOpen)

return None

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

def isBoardFull():
    return board.count(' ')<=1

def main():
global board
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 is None:
        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 _ in range [10]
    print ('-------')
    main()
  else:
    break

main()
```

Output :

```
 x |   | O
---+---+---
   |   |
---+---+---
   |   |
```

Please select a position to place an 'x' (1-9) : 5

```
 x |   | O
---+---+---
   | x |
---+---+---
   |   |
```

Computer placed an 'O' in position 9 :

```
 x |   | O
---+---+---
   | x |
---+---+---
   |   | O
```

Please select a position to place an 'x' (1-9) : 6

```
 x |   | O
---+---+---
   | x | x
---+---+---
   |   | O
```

Computer placed an 'O' in position 4 :

```
 x |   | O
---+---+---
 O | x | x
---+---+---
   |   | O
```

Please select a position to place an 'x' (1-9) : 8

```
 x |   | O
---+---+---
 O | x | x
---+---+---
   | x | O
```

Computer placed an 'O' in position 2 :

```
 x | O | O
---+---+---
 O | x | x
---+---+---
   | x | O
```

Please select a position to place an 'x' (1-9) : 7

```
 x | O | O
---+---+---
 O | x | x
---+---+---
 x | x | O
```

Tie game!

```
import random

# Initialize the game board board = [' ' for _ in range(10)]

def insertLetter(letter, pos): global board
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] # print(' | |')

def isWinner(bo, le): return (

(bo[7] == le and (bo[4] == le and (bo[1] == le and (bo[1] == le and (bo[2] == le and (bo[3]
== le and (bo[1] == le and (bo[3] == le and

def playerMove(): global board run = True while run:

+ ' | ' + board[8] + ' | ' + board[9])

)

bo[8] == le and bo[9] == le) or bo[5] == le and bo[6] == le) or bo[2] == le and bo[3] == le)
or bo[4] == le and bo[7] == le) or bo[5] == le and bo[8] == le) or bo[6] == le and bo[9] ==
le) or bo[5] == le and bo[9] == le) or bo[5] == le and bo[7] == le)

move = input('Please select a position to place an \'X\' (1-9): ') try:

move = int(move) if 1 <= move <= 9:

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 ValueError:

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

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

for let in ['O', 'X']:
for i in possibleMoves:

boardCopy = board[:] boardCopy[i] = let
if isWinner(boardCopy, let):

return i

cornersOpen = [i for i in possibleMoves if i in [1, 3, 7, 9]] if cornersOpen:

return selectRandom(cornersOpen)

if 5 in possibleMoves: return 5

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

return selectRandom(edgesOpen) return None # Indicates a tie

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

def isBoardFull(board): return board.count(' ') <= 1

def main():
global board
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 is None:

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 _ in range(10)] print('-----------------------------------') main()

else: break

Run the game

main()

OUTPUT:

```
-----------
   |   |
-----------
   |   |
Computer placed an 'O' in position 3 :
 X |   | O
-----------
   |   |
-----------
   |   |
Please select a position to place an 'X' (1-9): 5
 X |   | O
-----------
   | X |
-----------
   |   |
Computer placed an 'O' in position 9 :
 X |   | O
-----------
   | X |
-----------
   |   | O
Please select a position to place an 'X' (1-9): 6
 X |   | O
-----------
   | X | X
-----------
   |   | O
Computer placed an 'O' in position 4 :
 X |   | O
-----------
 O | X | X
-----------
   |   | O
Please select a position to place an 'X' (1-9): 8
 X |   | O
-----------
 O | X | X
-----------
   | X | O
Computer placed an 'O' in position 2 :
 X | O | O
-----------
 O | X | X
-----------
   | X | O
Please select a position to place an 'X' (1-9): 7
 X | O | O
-----------
 O | X | X
-----------
 X | X | O
Tie Game!
```

2. 8 Puzzle Breadth First Search Algorithm

LAB 3 - 8 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(states, visited_states):
  b = state.index(-1)
  d = []
  if b not in [0,1,2]:
    d.append('u')
  if b not in [6,7,8]:
    d.append('d')
```

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

    pos_moves_it_can = []

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

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

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

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

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

store 67

OUTPUT :

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

Pali
19/12/23

store
67

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

exp = []

    while len(queue) > 0:
        source = queue.pop(0)
        exp.append(source)
        print(source)
        if source==target:
            print("success")
            return

poss_moves_to_do = []
poss_moves_to_do = possible_moves(source,exp)

for move in poss_moves_to_do:

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

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

    #directions array

d = []
#Add all the possible directions

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

# If direction is possible then add state to move

pos_moves_it_can = []
# for all possible directions find the state if that move is played

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

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

    return [move_it_can for
in visited_states]
def gen(state, m, b):
```

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

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

move_it_can in pos_moves_it_can if move_it_can not

temp[b],temp[b+3]
temp[b],temp[b-3]
temp[b],temp[b-1]
temp[b],temp[b+1]

tested move to later check if "src == target"
```

OUTPUT:

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

3. 8 Puzzle Iterative Deepening Search Algorithm

LAB - 4 8 PUZZLE PROBLEM

```
class PuzzleNode:
    def __init__(self, state, parent = None, action = None):
        self.state = state
        self.parent = parent
        self.action = action

    def get_path(self):
        path = []
        current = self
        while current:
            path.append((current.state, current.action))
            current = current.parent
        return path[::-1]

def is_goal(state):
    goal_state = (1, 2, 3, 6, 4, 5, 0, 7, 8)
    return state == goal_state

def get_neighbors(state):
    neighbors = []
    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 = row + move[0], col + move[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            neighbor_state = list(state)
            neighbor_index = new_row * 3 + new_col
            neighbor_state[empty_index], neighbor_state[neighbor_index] = (neighbor_state[neighbor_index], neighbor_state
```

67

```
, [empty_index],
)
    neighbors.append(tuple(neighbor_state))
return neighbors

def depth_limited_search(node, goal_state, depth_limit):
  if is_goal(node.state):
    return True
  elif depth_limit == 0:
    return False
  else:
    for neighbor_state in get_neighbors(node.state):
        child = PuzzleNode(neighbor_state, node)
        if depth_limited_search(child, goal_state, depth_limit -1):
          return True
    return f False

if __name__ == "__main__":
   initial_state = (1, 2, 3, 0, 4, 5, 6, 7, 8)
   depth_limit = 1
   initial_node = PuzzleNode(initial_state)
   result = depth_limited_search(initial_node, (1,2,3,6,4,5,0,7,8),
                                 depth_limit)
   print(result)
```

OUTPUT:

dfs.py
True

store
67

```
class PuzzleNode:
def __init__(self, state, parent=None, action=None):

        self.state = state
        self.parent = parent
        self.action = action

def get_path(self): # Not required path = []

        current = self
        while current:

path.append((current.state, current.action))

            current = current.parent
        return path[::-1]

def is_goal(state):
goal_state = (1,2,3,6,4,5,0,7,8) return state == goal_state

def get_neighbors(state): neighbors = []

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 = row +
move[0], col + move[1] if 0 <= new_row < 3 and 0 <= new_col < 3:

neighbor_state = list(state)
neighbor_index = new_row * 3 + new_col neighbor_state[empty_index],
neighbor_state[neighbor_index] = (

neighbor_state[neighbor_index],

neighbor_state[empty_index], )

neighbors.append(tuple(neighbor_state)) return neighbors

def depth_limited_search(node, goal_state, depth_limit): if
is_goal(node.state):

        return True
    elif depth_limit == 0:
        return False
    else:

for neighbor_state in get_neighbors(node.state):
child = PuzzleNode(neighbor_state, node)
if depth_limited_search(child, goal_state, depth_limit - 1):

return True
        return False
```

```
if __name__ == "__main__":
initial_state = (1, 2, 3, 0, 4, 5, 6, 7, 8)
depth_limit = 1 # Set the depth limit as needed
initial_node = PuzzleNode(initial_state)
result = depth_limited_search(initial_node, (1,2,3,6,4,5,0,7,8),

depth_limit)
    print(result)
```

OUTPUT:

```
dfs.py
True
```

4. 8 Puzzle A* Search Algorithm

A* Algorithm :

```
import heapq

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

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

    def shuffle(self, puz, x1, y1, x2, y2):
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = 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
```

store 67

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

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

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

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

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

```
def process ( self , start_data , goal_data ) :
    start = Node ( start_data , 0, 0 )
    start = self.f( start, goal_data )
    self.open.append( start )
    print ( "\n\n" )

    while True :
        cur = self.open [0]

        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)
```

67

OUTPUT :

1 2 3
_ 4 6
7 5 8

1 2 3
4 _ 6
7 5 8

1 2 3
4 5 6
7 _ 8

1 2 3
4 5 6
7 8 _

20/12

Time Complexity of BFS : Worst case, it can be expressed as $O(b^d)$ where b is branching factor and d is depth of solution.

Time Complexity of A* : Depends on heuristic, if heuristic is perfect, then it is $O(d)$ and worst case $O(b^d)$

store 67

```
import heapq

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

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

def generate_child(self):
x, y = self.find(self.data, '_')
val_list = [[x, y-1], [x, y+1], [x-1, y], [x+1, y]] children = []
for i in val_list:

child = self.shuffle(self.data, x, y, i[0], i[1]) if child is not None:

child_node = Node(child, self.level+1, 0)

children.append(child_node) return children

def shuffle(self, puz, x1, y1, x2, y2):
if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp_puz = self.copy(puz)
temp = temp_puz[x2][y2] temp_puz[x2][y2] = temp_puz[x1][y1] temp_puz[x1][y1] = temp
return temp_puz

else:
return None

def copy(self, root): temp = []

for i in root: t = []

for j in i: t.append(j)

temp.append(t) return temp

def find(self, puz, x):
for i in range(0, len(self.data)):

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

return i, j

class Puzzle:
def __init__(self, size):

self.n = size self.open = [] self.closed = []

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

```
def h(self, start, goal): temp = 0

for i in range(0, self.n): for j in range(0, self.n):

if start[i][j] != goal[i][j] and start[i][j] != '_': temp += 1

return temp

def process(self, start_data, goal_data): 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("\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)

# Define puzzle states
start_state = [['1', '2', '3'], ['_', '4', '6'], ['7', '5', '8']] goal_state = [['1', '2', '3'], ['4', '5', '6'], ['7', '8',
'_']]

# Create Puzzle object and run the process

puz = Puzzle(3)
puz.process(start_state, goal_state)
```

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

1 2 3
4 _ 6
7 5 8

1 2 3
4 5 6
7 _ 8

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

5.Vacuum Cleaner

LAB 2 - VACCUM

```
def vaccum_world():
    goal_state = {'A':'0', 'B':'0'}
    cost = 0

    location_input = input("Enter location of Vaccum")
    status_input = input("Enter status of " + location_input)
    status_input_complement = input("Enter status of other room")
    print("Initial Location Condition" + str(goal_state))

    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
            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 cleaned.")
```

```
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 Cleared.")
else
    print ("No action" + str(cost))
    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'
        print ("COST for SUCK" + str(cost))
        print ("Location A has been cleared.")

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

    if status_input_complement == '1'
        print ("Location A is dirty.")
        print ("Moving LEFT to 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 cleared.")
  else:
    print ("No action" + str(cost))
    print ("Location A is already clean.")
    ~~print~~
print (" GOAL STATE: ")
print (goal_state)
print ("Performance Measurement: " + str(cost))

~~vacuu~~ vacuum_world()
```

OUTPUT :

```
Enter location of vacuum A
Enter status of A 1
Enter status of other room 1
Initial location condition {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A 1
Location A has been cleared.
Location B is dirty
Moving right to the Location B
COST for moving RIGHT 2
COST for SUCK 3
Location B has been cleared
GOAL STATE :
```

store 67

{'A' : '0', 'B' : '0'}
Performance Measurement : 3
Enter Location of Vacuum B
Enter status of B 0
Enter status of other room 1
Initial location condition {'A' : '0', 'B' : '0'}
Vacuum is placed in Location B
0
Location B is already clean.
Location A is dirty
Moving LEFT to the Location A
COST for moving LEFT 1
COST for SUCK 2
LOCATION A has been Cleaned.
GOAL STATE:
{'A' : '0', 'B' : '0'}
Performance Measurement : 2

```
def vacuum_world():
        # initializing goal_state

# 0 indicates Clean and 1 indicates Dirty

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

location_input = input("Enter Location of Vacuum") #user_input of location
vacuum is placed

status_input = input("Enter status of " + location_input) #user_input if
location is dirty or clean

status_input_complement = input("Enter status of other room")
print("Initial Location Condition" + str(goal_state))

if location_input == 'A':
# Location A is Dirty.
print("Vacuum is placed in Location A") if status_input == '1':

print("Location A is Dirty.")
# suck the dirt and mark it as clean goal_state['A'] = '0'
cost += 1 #cost for suck print("Cost for CLEANING A " + str(cost))
print("Location A has been Cleaned.")

if status_input_complement == '1': # if B is Dirty

print("Location B is Dirty.")
print("Moving right to the Location B. ")
cost += 1 #cost for moving right print("COST for moving RIGHT" +
str(cost))
# suck the dirt and mark it as clean
goal_state['B'] = '0'
cost += 1 #cost for suck print("COST for SUCK " + str(cost))
print("Location B has been Cleaned. ")

else:
print("No action" + str(cost))
# suck and mark clean print("Location B is already clean.")

if status_input == '0':
print("Location A is already clean ")
if status_input_complement == '1':# if B is Dirty

print("Location B is Dirty.")
print("Moving RIGHT to the Location B. ")
cost += 1 #cost for moving right print("COST for moving RIGHT " +
str(cost))
# suck the dirt and mark it as clean
goal_state['B'] = '0'

cost += 1 #cost for suck print("Cost for SUCK" + str(cost))
print("Location B has been Cleaned. ")
```

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

else:
print("Vacuum is placed in location B") # Location B is Dirty.
if status_input == '1':

print("Location B is Dirty.")
# suck the dirt and mark it as clean goal_state['B'] = '0'
cost += 1 # cost for suck print("COST for CLEANING " print("Location B has
been

if status_input_complement # if A is Dirty

== '1':

print("Location A is Dirty.") print("Moving LEFT to the Location A. ")
cost += 1 # cost for moving right print("COST for moving LEFT" +
str(cost)) # suck the dirt and mark it as clean goal_state['A'] = '0'
cost += 1 # cost for suck
print("COST for SUCK " + str(cost)) print("Location A has been Cleaned.")

else:
    print(cost)
    # suck and mark clean

print("Location B is already clean.")

if status_input_complement == '1': # if A is Dirty print("Location A is
Dirty.")
print("Moving LEFT to the Location A. ")
cost += 1 # cost for moving right

print("COST for moving LEFT " + str(cost)) # suck the dirt and mark it as
clean goal_state['A'] = '0'
cost += 1 # cost for suck

print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ") else:

print("No action " + str(cost))

+ str(cost))
Cleaned.")
                # suck and mark clean

print("Location A is already clean.")

    # done cleaning
```

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

vacuum_world()
```

OUTPUT:

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

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

6. Knowledge Base Entailment

AI LAB - 6

1)

```
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
```

store
67

```
def main():
    generate_truth_table()

    if query_entails_knowledge():
        print("\n knowledge base entails query")
    else:
        print("\n knowledge base does not entail query")

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

OUTPUT :

q	p	r	Expression (KB)	Query (r)
True	True	True	False	True
True	True	False	False	False
True	False	True	False	True
True	False	False	False	False
False	True	True	False	True
False	True	False	False	False
False	False	True	False	True
False	False	False	False	False

Knowledge Base entails query

2)

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

          if expression_result and not query_result:
          return false

      return true

def main():
  generate_truth_table()

  if query_entails_knowledge():
    print ("\n Knowledge base entails query");
  else:
    print ("\nKnowledge base does not entail query");
```

slrre
67

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

Output :

q	p	r	kb	Query
True	True	True	True	True
True	True	False	True	False
True	False	True	False	False
True	False	False	True	False
False	True	True	True	True
False	True	False	True	False
False	False	True	False	False
False	False	False	False	False

Knowledge Base does not entail query.

store 67

1) Evaluate the given expression (~q v ~p v r) ^ (~q ^ p) ^ q . Check whether knowledge base entails query or not.

def evaluate_expression(q, p, r):
# Evaluate the given expression (~q v ~p v r) ^ (~q ^ p) ^ q expression_result = ((not q or not p or r) and (not q and p) and q) return expression_result

def generate_truth_table():
# Print the header of the truth table
print(" q | p | r | Expression (KB) | Query (r)") print("---|---|---|-----------------|------------")

Evaluate and print each row of the truth table 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():
# Check if query entails the 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 the expression is true and the query is false, query does not entail the knowledge

if expression_result and not query_result: return False

If the loop completes without returning, query entails the knowledge

return True

def main():
# Generate and print the truth table generate_truth_table()

Check if query entails the knowledge and print the result if query_entails_knowledge():

print("\nKnowledge base entails query") else:

print("\nKnowledge base does not entail query")

if __name__ == "__main__": main()

input

```
q | p | r | Expression (KB) | Query (r)
---|---|---|-----------------|------------
 True | True | True | False              | True
 True | True | False | False             | False
 True | False | True | False             | True
 True | False | False | False            | False
 False | True | True | False             | True
 False | True | False | False            | False
 False | False | True | False            | True
 False | False | False | False           | False

Knowledge base entails query
```

2) Evaluate the given expression (p v q) ^ (~r ^ p) . Check whether knowledge base entails query or not.

```
def evaluate_expression(q, p, r):
# Evaluate the given expression (p v q) ^ (~r ^ p) expression_result = ((p or q) and (not r or p)) return expression_result

def generate_truth_table():
# Print the header of the truth table
print(" q | p | r | Expression (KB) | Query (r)") print("---|---|---|-----------------|------------")

# Evaluate and print each row of the truth table for q in [True, False]:

for p in [True, False]: for r in [True, False]:

expression_result = evaluate_expression(q, p, r) query_result = p and r

print(f" {q} | {p} | {r} | {expression_result} | {query_result}")

def query_entails_knowledge():
# Check if query entails the 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 = p and r

# If the expression is true and the query is false, query does not entail the knowledge

if expression_result and not query_result: return False

# If the loop completes without returning, query entails the knowledge

return True
```

def main():
# Generate and print the truth table generate_truth_table()

Check if query entails the knowledge and print the result if query_entails_knowledge():

print("\nKnowledge base entails query") else:

print("\nKnowledge base does not entail query")

if __name__ == "__main__":

main()

input

```
 q | p | r | Expression (KB) | Query (r)
---|---|---|-----------------|------------
 True | True | True | True                | True
 True | True | False | True               | False
 True | False | True | False              | False
 True | False | False | True              | False
 False | True | True | True               | True
 False | True | False | True              | False
 False | False | True | False             | False
 False | False | False | False            | False

Knowledge base does not entail query
```

7. Knowledge Base Resolution

WEEK - 7

Create a kb using PL and Prove the query using resolution.

```
import re
def main(rules, goals)
  rules = rules.split('')
  steps = resolve(rules, goal)
  print('\n Step\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 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 contradict(goal, clause):
  contradiction = [f'{goal}v{negate(goal)}',
                   f'{negate(goal)}v{goal}']
  return clause in contradiction or reverse(clause) in contradiction
```

```
def resolve(rules, goal):
    temp = rules.copy()
    temp += [negate(goal)]
    steps = dict()

    for rule in temp:
        steps[rule] = 'given'

    steps[negate(goal)] = '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 terms:
                    t1 = [t for t in terms1 if t != c]
                    t2 = [t for t in terms2 if t != negate]
                    gen = t1 + t2
                    if len(gen) == 2:
                        if gen[0] != negate(gen(1)):
                            clauses += [f'{gen[0]} v {gen[1]}']
                    else:
                        if contradiction(goal, 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 the
                            term null
```

```
        elif len(gen) = 1:
            clauses+ = [f'gen[0]}']

else:
    if contradiction (goal, f'{ terms1 [0]} v{ terms2[0]}'):
    temp. append (f'{ terms1 [0] v{ terms 2[0]')
    steps['] = f"Resolved { temp(i)} and { temp (j)} to
        { temp [-1]} which is in turn null. \n A
        contradiction is found when { gal (goal)} is
        assumed as true. Hence, { goal} 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

rules = 'RV~P RV~Q ~RVP ~RVQ'
goal = 'R'
main (rules, goal)
```

Output:

Step	Clause	Derivation
1.	R V ~P	given
2.	R V ~Q	given
3.	~R V P	given

4. ~R v Q Given

5. ~R Negated conclusion

~~R~~ Resolved R v ~P and ~R v P to R v ~R which is in turn null

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

Pall 10/1/24.

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

```
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 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

split_terms('~PvR')
['~P', 'R']
def contradiction(goal, clause):

contradictions = [ f'{goal}v{negate(goal)}', f'{negate(goal)}v{goal}']

return clause in contradictions or reverse(clause) in contradictions def resolve(rules, goal):

temp = rules.copy() temp += [negate(goal)] steps = dict()

for rule in temp: steps[rule] = 'Given.'

steps[negate(goal)] = '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(goal,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(goal)} is assumed as true. Hence, {goal} is true."

return steps elif len(gen) == 1:

clauses += [f'{gen[0]}'] else:

if contradiction(goal,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(goal)} is assumed as

true. Hence, {goal} 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

rules = 'Rv~P Rv~Q ~RvP ~RvQ' #(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)
goal = 'R'
main(rules, goal)

rules = 'PvQ PvR ~PvR RvS Rv~Q ~Sv~Q' # (P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)
main(rules, 'R')
```

input

```
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.

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

8. Unification

WEEK: 8

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

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

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

  substitution = { }

  for a1, a2 in zip(args1, args2):
     if a1.islower() and a2.islower() and a1 != a2:
        substitution[a1] = a2
     elif a1.islower() and not a2.islower():
        substitution[a1] = a2
     elif not a1.islower() and a2.islower():
        substitution[a2] = a1
     elif a1 != a2
        print("Expressions cannot be unified. Incompatible
     arguments.")
        return None
  return substitution

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

store
67

```
        return expr

if __name__ == "__main__":
    expr1 = input("Enter the first expression: ")
    expr2 = input("Enter the second expression: ")
    substitution = 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 expression 1: {expr1_result}')
        print(f'Unified expression 2: {expr2_result}')
```

<u>Outputs:</u>

Enter the first expression: knows(f(x), y)
Enter the second expression: knows(J, John)
The substitutions are
f(x) / J
y / John
Unified expression 1: knows(J, John)
Unified expression 2: knows(J, John)

Enter the first expression: Student(x)
Enter the second expression: Teacher(Rose)
Expressions cannot be unified. Different functions.

store
67

Enter the first expression : knows(John, x)
Enter the second expression : knows(y, Mother(y))
The substitutions are :
y / John
x / Mother (y
Unified expression 1 : knows (John, Mother (y)
Unified expression 2 : knows (John, Mother (John))

Enter the first expression : Like (A, y)
Enter the second expression : Like (K, g(x))
Expressions cannot be unified. Incompatible arguments.

Pall 18/1/24.

```
def unify(expr1, expr2):
# Split expressions into function and arguments func1, args1 =
expr1.split('(', 1)
func2, args2 = expr2.split('(', 1)

    # Check if functions are the same
    if func1 != func2:
print("Expressions cannot be unified. Different functions.") return
None

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

substitution = {}
# Unify arguments

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():

substitution[a1] = a2
elif not a1.islower() and a2.islower():

        substitution[a2] = a1
    elif a1 != a2:

print("Expressions cannot be unified. Incompatible

arguments.")
    return substitution

def apply_substitution(expr, substitution):

return None
for key, value in substitution.items(): expr = expr.replace(key,
value)

return expr

# Main program
if __name__ == "__main__":
    # Sample input

expr1 = input("Enter the first expression: ") expr2 = input("Enter
the second expression: ")
# Unify expressions

substitution = unify(expr1, expr2)

# Display result
```

```
if substitution:
print("The substitutions are:")
for key, value in substitution.items():

        print(f'{key} / {value}')

# Apply substitution to original expressions

expr1_result = apply_substitution(expr1, substitution) expr2_result
= apply_substitution(expr2, substitution)
print(f'Unified expression 1: {expr1_result}') print(f'Unified
expression 2: {expr2_result}')
```

Outputs:

```
Enter the first expression: knows(f(x),y)
Enter the second expression: knows(J,John)
The substitutions are:
f(x) / J
y / John
Unified expression 1: knows(J,John)
Unified expression 2: knows(J,John)
```

```
Enter the first expression: Student(x)
Enter the second expression: Teacher(Rose)
Expressions cannot be unified. Different functions.
```

```
Enter the first expression: knows(John,x)
Enter the second expression: knows(y,Mother(y))
The substitutions are:
y / John
x / Mother(y
Unified expression 1: knows(John,Mother(y)
Unified expression 2: knows(John,Mother(John))
```

```
Enter the first expression: like(A,y)
Enter the second expression: like(K,g(x))
Expressions cannot be unified. Incompatible arguments.
```

9. FOL to CNF

WEEK - 9

FOL to CNF

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

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

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~','')
    flag = '[' in string
    string = string.replace('~[','')
    string = string.strip(']')
    for predicate in getPredicates(string)
        string = string.replace(predicate, f'~{predicate}')
    s = list(string)
    for i, c in enumerate(string):
        if c == '|':
            s[i] = '&'
        elif c == '&'
            s[i] = '|'
    string = ''.join(s)
    string = string.replace('~~','')
    return f'[{string}]' if flag else string

def Skolemization(sentence):
    SKOLEM_CONSTRAINTS = [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_CONSTA
               NTS.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_CONST
               ANTS.pop(0)}({aL[0] if len(aL) else match[1]
               })')
     return statement

import re
def fol_to_cnf(fol):
  statement = fol.replace("<=>", "_")
  while '_' in statement:
     i = statement.index('_')
     new_statement = '[' + statement[:i] + '=>' + statement[i+1:] +
       ']&[' + statement[i+1:] + '=>' + statement[:i] + ']'
         statement = new_statement
     statement = statement.replace("=>", "-")
     expr = '\[([^]]+)\]'
     statements = re.findall(expr, statement)
```

```
for i, s in enumerate(statements):
    if '[' in s and ']' not in s:
        statements[i] += ']'
for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
while '_' in statement:
    i = statement.index('_')
    br = statement.index('[') if '[' in statement else 0
    new_statement = '~' + statement[br:i] + '|' + statement[i
        +1:]
    statement = statement[:br] + new_statement if br > 0 else
        new_statement
while '~∀' in statement:
    i = statement.index('~∀')
    statement = list(statement)
    statement[i], statement[i+1], statement[i+2] = '∃',
    statement[i+2], '~'
    statement = ''.join(statement)
while '~∃' in statement:
    i = statement.index('~∃')
    s = list(statement)
    s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
    statement = ''.join(s)
statement = statement.replace('~[∀', '[~∀')
statement = statement.replace('~[∃', '[~∃')
expr = '(~[∀|∃].)'
statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, fol_to_cnf(s))
expr = '(~[∀|∃].)'
statements = re.findall(expr, statement)
for s in statements:
```

```
    statement = statement.replace(s, fol_to_cnf(s))
expr = '~\[[^]]+\]'
statements = re.findall(expr, statement)
for s in statements:
    statement = statement.replace(s, DeMorgan(s))
return statement
print(Skolemization(fol_to_cnf("∀x food(x) => likes(John, x)")))
print(Skolemization(fol_to_cnf("∀x[∃z[loves(x, z)]]")))
print(fol_to_cnf("[american(x)&weapon(y)&sells(x, y, z)&
hostile(z)] => criminal(x)"))
```

Output:

```
~food(A) | likes(John, A)
[loves(x, B(x))]
[~american(x) | ~weapon(y) | ~sells(x, y, z) | ~hostile(z)] |
    criminal(x)
```

store
67

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

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

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

def DeMorgan(sentence):
string = ''.join(list(sentence).copy()) string =
string.replace('~~','')
flag = '[' in string
string = string.replace('~[','')

string = string.strip(']')
for predicate in getPredicates(string):

string = string.replace(predicate, f'~{predicate}') s = list(string)
for i, c in enumerate(string):

    if c == '|':
        s[i] = '&'
    elif c == '&':
        s[i] = '|'

string = ''.join(s)
string = string.replace('~~','')
return f'[{string}]' if flag else string
def Skolemization(sentence):
SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'),

ord('Z')+1)]
statement = ''.join(list(sentence).copy()) matches =
re.findall('[∀∃].', statement) for match in matches[::-1]:

statement = statement.replace(match, '') statements =
re.findall('\[\[[^]]+\]]', statement) for s in statements:

statement = statement.replace(s, s[1:-1]) for predicate in
getPredicates(statement):

attributes = getAttributes(predicate)
if ''.join(attributes).islower():
                statement =

statement.replace(match[1],SKOLEM_CONSTANTS.pop(0)) else:
aL = [a for a in attributes if a.islower()]
aU = [a for a in attributes if not a.islower()][0] statement =
statement.replace(aU,
```

```
f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')
return statement

import re
def fol_to_cnf(fol):

statement = fol.replace("<=>", "_") while '_' in statement:

        i = statement.index('_')

new_statement = '[' + statement[:i] + '=>' + statement[i+1:] +
']&['+ statement[i+1:] + '=>' + statement[:i] + ']'

        statement = new_statement
statement = statement.replace("=>", "-") expr = '\[([^]]+)\]'
statements = re.findall(expr, statement) for i, s in
enumerate(statements):

if '[' in s and ']' not in s: statements[i] += ']'

for s in statements:
statement = statement.replace(s, fol_to_cnf(s))

while '-' in statement:
i = statement.index('-')
br = statement.index('[') if '[' in statement else 0 new_statement =
'~' + statement[br:i] + '|' + statement[i+1:] statement =
statement[:br] + new_statement if br > 0 else

new_statement
while '~∀' in statement:

i = statement.index('~∀')
statement = list(statement)
statement[i], statement[i+1], statement[i+2] = '∃',
statement[i+2], '~'
        statement = ''.join(statement)

while '~∃' in statement:
i = statement.index('~∃')
s = list(statement)
s[i], s[i+1], s[i+2] = '∀', s[i+2], '~' statement = ''.join(s)

statement = statement.replace('~[∀','[~∀') statement =
statement.replace('~[∃','[~∃') expr = '(~[∀|∃].)'
statements = re.findall(expr, statement)
for s in statements:
statement = statement.replace(s, fol_to_cnf(s))

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

```
for s in statements:
statement = statement.replace(s, DeMorgan(s))

return statement
```

Output:

```
[12] print(Skolemization(fol_to_cnf("∀x food(x) => likes(John, x)")))

     ~ food(A) | likes(John, A)

[13] print(Skolemization(fol_to_cnf("∀x[∃z[loves(x,z)]]")))

     [loves(x,B(x))]

     print(fol_to_cnf("[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)"))

     [~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)
```

10. Forward reasoning

WEEK 10

Prove query using Forward Reasoning

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

def getAttributes(string):
  expr = '\([^)]+\)'
  return 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
```

67

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

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

  def evaluate(self, facts):
    constants = {}
    new_lhs = []
    for fact in facts:
      for val in self.lhs
        if val.predicate == fact.predicate:
          for i, v in enumerate(val.getVariables()):
            if v:
              constants[v] = fact.getConstants()[i]
          new lhs.append(fact)
        predicate, attributes = getPredicates(self.rhs.expression)[0],
  str(getAttributes(self.rhs.expression)[0])
      for key in constants:
```

```
            if constants[key]:
                attributes = attributes.replace(key, constants[key])
        expr = f'{predicate}{attributes}'
        return Fact(expr) if len(new_lhs) and all([f.getResult()
            for f in new_lhs]) else None

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

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

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

def display(self):
    print("All facts:")
    for i, f in enumerate(set([f.expression for f in self.facts])):
```

```
print(f'\t{i+1}. {f}')

kb= KB()
kb.tell('missile(x) => weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x, America) => hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono, America)')
kb.tell('owns(Nono, M1)')
kb.tell('missile(x) & owns(Nono, x) => sells(West, x, Nono)')
kb.tell('american(x) & weapon(y) & sells(x, y, z) & hostile(z) => criminal(x)')
kb.query('criminal(x)')
kb.display()
```

Output:

Querying criminal(x):
1. criminal(West)

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

1/2/24

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

```
import re
def isVariable(x):

return len(x) == 1 and x.islower() and x.isalpha() def
getAttributes(string):
expr = '\([^)]+\)'
matches = re.findall(expr, string) return matches

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

class Fact:
def __init__(self, expression):

self.expression = expression
predicate, params = self.splitExpression(expression) self.predicate
= predicate
self.params = params
self.result = any(self.getConstants())

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

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

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

def substitute(self, constants):
c = constants.copy()
f = f"{self.predicate}({','.join([constants.pop(0) if

isVariable(p) else p for p in self.params])})" return Fact(f)
class Implication:
def __init__(self, expression):

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

    def evaluate(self, facts):
        constants = {}
        new_lhs = []
        for fact in facts:
```

```
for val in self.lhs:
if val.predicate == fact.predicate:
for i, v in enumerate(val.getVariables()): if v:

constants[v] = fact.getConstants()[i] new_lhs.append(fact)

predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])

        for key in constants:
            if constants[key]:

attributes = attributes.replace(key, constants[key]) expr =
f'{predicate}{attributes}'
return Fact(expr) if len(new_lhs) and all([f.getResult() for f

in new_lhs]) else None
class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()
    def tell(self, e):
        if '=>' in e:

self.implications.add(Implication(e)) else:

            self.facts.add(Fact(e))
        for i in self.implications:
            res = i.evaluate(self.facts)
        if res:
            self.facts.add(res)

def query(self, e):
facts = set([f.expression for f in self.facts]) i=1
print(f'Querying {e}:')
for f in facts:

if Fact(f).predicate == Fact(e).predicate: print(f'\t{i}. {f}')
```

```
i += 1

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}')
```

Output:

```
kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono,America)')
kb.tell('owns(Nono,M1)')
kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

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