# TIC TAC 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("  |  |  ")
    print(board[1] + " | " + board[2] + " | " +
          board[3])
    print("  |  |  ")
    print("-----------")
    print(" " + board[4] + " | " + board[5] +
          " | " + board[6])
    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():
   global board
   run = True
   while run:
      move = input("select Xpos(1-9)")
   try:
      move = int(move)
      if 1<=move<=9:
         if spaceIsFree(move):
            run=False
            insertLetter('X', move)
         else:
            print("occupied")
      else:
         print("type no. in range")
   except:
      print("Is type 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
```

Scanned with OKEN Scanner

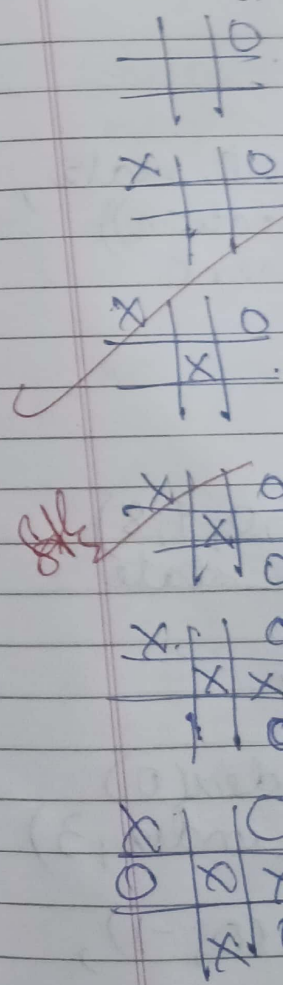
Vaishnavikannath

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

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


def main():
  global board
  printBoard(board)

while not isBoardFull(board):
    if not isWinner(board,'O'):
      playerMove()
      printBoard(board)
    else:
      print("O won")
      break

  if not isWinner(board,'X'):
    move=compMove()
    if move is None:
      print("Tie game")
  else:
    print('X's won')
    break

if isBoardFull(board):
```

print("Tie

while True
answer
if answ
board
print
main()
else
break

main()

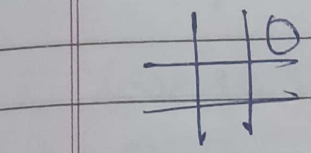


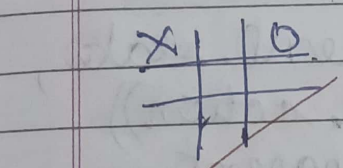
Scanned with OKEN Scanner

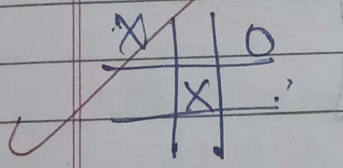
print('Tie')

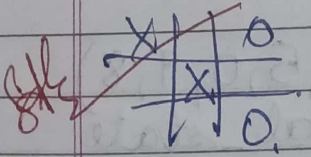
while True:
answer = input('Do you play to.')
if answer.lower() == 'y' or
board = [" for in range(10)]
print(' - - - ')
main()
else
break

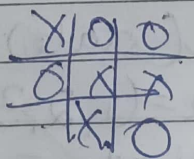
main()

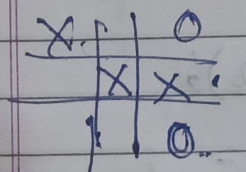


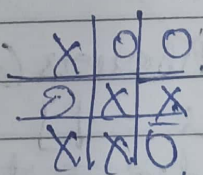


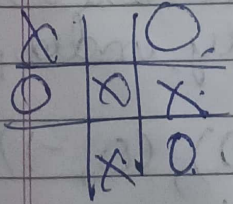












Tie game.

# Vacuum World

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

    location_ip = input("Enter vac loc")
    status_ip = input("status" + location_ip)
    status_ip_comp = input("status, c")
    print("Initial loc cond" + str(goal_
          state))

    if location_ip == 'A':
        print("In A")
        if status_ip == 1:
            print("A dirty")
            goal_state['A'] = 0
            cost += 1
            print("cost" + str(cost))
            print("A cleaned")

    if status_ip_comp == '1':
        print("B dirty")
        print("moving to B")
        cost += 1
        print("cost it" + str(cost))
        goal_state['B'] = '0'
        cost += 1
        print("COST for suck" + str(cost))
        print("B cleaned")
    else:
        print("B already clean")
```

```
if status_input == '0':
    print("A clean")
    if status_ip_comp == '1':
    print("B dirty, moving at")
    cost += 1
    print("cost: " + str(cost))
    goal_state['B'] = 0
    cost += 1
    print("suck cost" + str(cost))
    print("B clean")
else:
    print("NO action, B clean")

else:
    print("In B")
    if status_ip == '1':
    print("B dirty")
    goal_state['B'] = '0'
    cost += 1
    print("cost" + str(cost))
    print("B clean")

    if status_ip_comp == '1':
    print("A dirty going TO A")
    cost += 1
    print(str(cost))
    goal_state['A'] = '0'
    cost += 1
    print("cost for suck " + str(cost))
    print("Now A clean")
else:
    print(cost)
    print("B clean")
```

Scanned with OKEN Scanner

```
if status_ip_comp == '1':
    print("A dirty")
    print("Moving left")
    cost += 1
    goal_state['A'] = '0'
    cost += 1
    print(str(cost))
    print("A cleaned")

else:
    print("No action")
    print("A already clean")

print("Goal state")
print(goal_state)
print("Perf: " + str(cost))

vacuum_world()
```

Enter location of vacuum
Status of A
Status of B
for
in A
loc A is dirty
Cost of cleaning A is 1
A is clean
B is dirty
Moving to B
cost of mov. = 3
cost of cleaning B 1
B clean
Goal state
{A: 0, B: 0}

Scanned with OKEN Scanner

# 8 puzzle problem

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

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 moves in poss_moves_to_do:

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

def possible_moves(state, visited_states)
```

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

poss_moves_it_can=[]

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

return [move_it_can for move_it_can
in poss_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],temp[b-3] = temp[b-3],
                        temp[b]

  if m=='r':
    temp[b],temp[b+1] = temp[b+1],
                        temp[b]
```

```
if m == 'l':
    temp[b], temp[b-1] = temp[b-1], temp[b]
return temp
```

OUTPUT:

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

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

8 puzzle problem using
Depth limit search

```
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_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)]:
```

Scanned with OKEN Scanner

(1,0)
(0,1)

new_row, new_col = row + move[0],
col + move[1]

if 0<=new_row<3 and
0<=new_col<3:

neighbour_state = list(state)
neighbour_index = new_row*3 +
new_col.

neighbour_state[empty_index],
neighbour_state[neighbour_index] =
(neighbour_state[neighbour_index],
neighbour_state[empty_index],).

neighbours.append(tuple(neighbour
_state)).

return neighbours.

def dls(node, goal_state,
depth_limit):

if is_goal(node.state):
return True.

elif depth_limit==0:
return False.

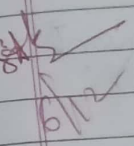
else:

Scanned with OKEN Scanner

```
for neighbour_state in get_neighbours(node.state):
    child := PuzzleNode(neighbour_state, node)
    if dls(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
    initial_node = PuzzleNode(initial_state)
    result = dls(initial_node, (1,2,3,6,4,5,0,7,8), depth_limit)
    print(result)
```

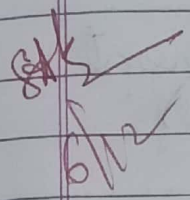
OUTPUT:

initial_state = (1,2,3,0,4,
target = (1,2,3,6,4,5,0,
depth_limit = 1
True



OUTPUT:

initial_state = (1, 2, 3, 0, 4, 5, 6, 7, 8)
target = (1, 2, 3, 6, 4, 5, 0, 7, 8)
depth_limit = 1
True



Scanned with OKEN Scanner

* Implement 8 puzzle problem using greedy BFS.

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent
                 =None):
        self.state = state
        self.parent = parent
        self.cost = 0

    def __lt__(self, other):
        return self.cost < other.cost

def mh(state, goal_state):
    dis = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != g_s[i][j]:
                x, y = divmod(goal_state[i][j], 3
                dis += abs(x-i) + abs(y-j)
    return dis

def get_blank_pos(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
def get_neighbours(node):
    i, j = get_blank_position(node.state)
    neigh = []

    for x, y in ((i+1, j), (i-1, j), (i, j+1),
                 (i, j-1)):

        if 0<=x<3 and 0<=y<3:
            neig = [row[:] for row in
                    node.state]
            neig[i][j], neig[x][y] = neig[x][y],
                                     neig[i][j]

            neigh.append(PuzzleNode(neig, parent
                                    =Node))

    return neigh

def gbfs(initial_state, g_s, heu):
    in = PuzzleNode(initial_state)
    gs = PuzzleNode(goal_state)

    if initial_state == goal_state:
        return [initial_state]

    priority_queue = [initial_node]
    visited_states = set()

    while prio_queue:
        current_node = heapq.heappop
                       (prio_queue)
```

```
if current_node.state == goal_state:
    path = [current_node.state]
    while current_node.parent:
        current_node = current_node.parent
        path.append(current_node.state)
    path.reverse()
    return path

visited_states.add(tuple(map(tuple, current_node.state)))

for neighbour in neighbours:

if tuple(map(tuple, neighbour.state)) not in visited_states:
    neighbor, cost = heuristic(
        neighbour.state, goal_state)

heapq.heappush(priority_queue, neighbour)

return None

initial_state = [
        [1, 2, 3]
        [4, 0, 5]
        [6, 7, 8] ]
goal_state = [
        [1, 2, 3]
        [4, 5, 6]
        [7, 8, 0] ]
```

Scanned with OKEN Scanner

path = gbfs(initial_state, goal_state, heuristic)

step 0
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

step 1
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

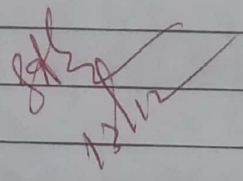
step 2
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

A star

```
import heapq.

class PuzzleNode
  def __init__(self, state, parent
                =None, g=0, h=0)

    self.state = state
    self.parent = parent
    self.g = g
    self.h = h
    self.f = self.g + self.h

  def __lt__(self, other):
      return self.f < other.f
```

sample o/p:

Reached in 1 move



# TABLE FILLING

```
def eval_first(premise, conc):
    models = [
    {'p': False, 'q': False, 'r': False
     'p': False, 'q': False, 'r': True
     'p': False, 'q': True, 'r': False
     'p': False, 'q': True, 'r': True
     'p': True, 'q': False, 'r': False
     'p': True, 'q': False, 'r': True
     'p': True, 'q': True, 'r': False
     'p': True, 'q': True, 'r': True }]

    entails = True
    for model in models:
        if eval_exp(premise, model) !=
eval_exp(conc, model):
            entails = False
            break

    return entails

def eval_sec(premise, concl):
    models = [
{'p': p, 'q': q, 'r': r}
for p in [True, False]
for q in [True, False]
for r in [True, False]
]
entails = all(evaluate_exp(premise,
                    model) for
model in models if eval_exp.
(conclusion, model) and
evaluate_exp(premise, model)
```

```
return entails

def evaluate_exp(exp, model):
    if isinstance(exp, str):
        return model.get(expression)

    elif isinstance(expression, tuple):
        op = expression[0]
    if op == 'not'
        return not evaluate_exp(
            exp[1], model)

    elif op == 'and'
        return evaluate_exp('exp[1],
            model) and

        evaluate_exp (exp[2], model)
    elif op == 'if'
        return (not evaluate_exp (exp[1],
        model)) or evaluate_exp('exp[2],
            model)

    elif op == 'θ':
        return eval_exp (exp[1])
        model) or eval_exp (exp[2],
                model)

first_prem = ['and', ('or', 'p', 'q'),
('or', ('not', 'r'), 'p')]
first_conc = ('and', 'p', 'r')
second_prem = ('and', ('or', ('not'
'q'), 'r'), ('and', ('or', ('not', 'q'),
('and', ('not', 'q'), ('p', 'q')
```

```
second_conc = 'r'
result = first = e
        (
    first conclusion
- result second
    second

if result_secon
    print("2 p

else:
    print('2 pr

if result_fi
    print('1: P
else:
    print("
    conc

O/P
1: Premise do
2: Premise en
```

second_concl = 'r'

result_first = evaluate_first
(first_premise,
first conclusion)

result_second = evaluate_second(
second premise, second concl)

if result_second:
print("2 premise entails concl")

else:
print('2: premise does not entail
concl")

if result_first:
print(1: Prem entail concl").

else:
print("1st Prem does not entail
concl")

O/P

1: Premise does not entail concl
2: Premise entails concl.

27/12

# KB Enrollment

```
import re

def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\n\nStep\tClause\tDerivation\t')
    print('-'*30)
    i = 1
    for step in steps:
        print(f'{i}.\t{step}\t\t'
              f'{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')

def contr(goal, clause):
    contr = [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 conc.'
    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 t1 != 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]}'):
```

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 conc'
i=0
while i<len(temp):
  n = len(temp)
  j = (i+1) % n
  clauses = []
  while j!=i:
    terms1 = split_terms(terms[i])
    terms2 = split_terms(temp[i])
    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.

return steps
elif len(gen) == 1:
clauses += [f'{gen[0]}']
else:
if contradiction(goal, f'{terms[0]} v {terms2[0]}'):

steps[' '] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]} which is in turn null. \n A contradiction is found when {negate(goal)} is assumed is 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 = 'R V ~P  R V ~Q  ~R V P  ~R V Q'
goal = 'R'
main(rules, goal)

1. R V ~P  Given
   R V ~Q  Given
   ~R V P  Given
   ~R V Q  Given
   ~R  Negated conclusion
   - Resolved R V ~P to R V ~R which is in turn null.

10/1/24

FOL (unification)

```
import re
def getAttr(exp):
    exp = exp.split("(")[1:]
    exp = "(".join(exp)
    exp = exp[:-1]
    exp = re.split("(?<!\(.),(?!.\))",
                   exp)
    return exp

def getInitialPred(exp):
    return exp.split("(")[0]

def isConstant(char)
    return char.isupper() and len(char)==1

def isVariable(char)
    return char.islower and len(char)==1

def replaceAttr(exp, old, new):
    attrs = getAttr(exp)
    for index, val in enumerate(attrs):
        if val == old
            attributes[index] = new predicate
                = getInitialPred(exp)

    return predicate + "(" + ",".join(attrs) +
                ")"
```

```
def apply(exp, subs):
    for sub in subs
    new, old = sub
    exp = replaceAttr(ex
    return exp

def checkOccurs(var
    if exp.find(var
    return False
    return True

def getFirstPart(e
    attr = getAttr(e
    return attr[

def getRemPart
    pred = getIn
    attr = getAtt
    newExp = pr

    return new

def unify(ex
    if exp1 == ex
        return[
    if isConst
        if exp1
            retu

    if isCons
        return
    if isConst
        retur
```

Scanned with OKEN Scanner

```
def apply(exp, subs):
    for sub in subs:
        new, old = sub
        exp = replaceAth(exp, old, new)
    return exp

def checkOccurs(var, exp):
    if exp.find(var) == -1:
        return False
    return True

def getFirstPart(exp):
    ath = getAttrs(exp)
    return ath[0]

def getRestPart(exp):
    pred = getInitialPred(exp)
    ath = getAths(exp)
    newExp = pred + "(" + ",".join(ath[1:]) + ")"
    return newExp

def unify(exp1, exp2):
    if exp1 == exp2:
        return []
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            return False
    if isConstant(exp1):
        return [(exp1, exp2)]
    if isConstant(exp2):
        return [(exp2, exp1)]
```

Scanned with OKEN Scanner

```
if isVariable(exp1):
    if checkOccurs(exp1, exp2):
        return False
    else:
        return [(exp2, exp1)]

if isVariable(exp2):
    if checkOccurs(exp2, exp1):
        return False
    else
        return [(exp1, exp2)]

if getInitPred(exp1) != getInitPred(exp2):
    print("Can't unify")
    return False

attrCount1 = len(getAttributes(exp1))
attrCount2 = len(getAttr(exp2))
if attr1 != attr2:
    return False

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

initSub = unify(head1, head2)
if not initSub:
    return False
if attrCnt1 == 1:
    return initSub

tail1 = getRemPart(exp1)
tail2 = getRemPart(exp2)
```

```
if initialSub != []:
    tail1 = apply(tail1, initialSub)
    tail2 = apply(tail2, initialSub)

remSub = unify(tail1, tail2)

if not remSub:
    return False
initSub.extend(remSub)
return initSub

exp1 = "knows(A, x)"
exp2 = "knows(y, mother(y))"
subs = unify(exp1, exp2)
print(subs)
```

Scanned with OKEN Scanner

if initialSub != []
    tail1 = apply(tail1, initSub)
    tail2 = apply(tail2, initSub)

remSub = unify(tail1, tail2)

if not remSub:
    return False
initSub.extend(remSub)
return initSub

exp1 = "knows(A, x)"
exp2 = "knows(y, y)"
subs = unify(exp1, exp2)
print(subs)

Scanned with OKEN Scanner

[a]?

greedy & Non greedy match

<.x> {arb}{c}

↳ Full match

<.x?.> → Non greedy match {a}

x+, ++, ?+ (possessive quantifiers)

a*a aaaa

↳ Matches all 4 as then is backtracks thus matching 3 a's.

a*{3}+a

{m} copies of prev RE be matched

a{3,5} → 3 to 5 as
m n
omitting m → OLB
" n → UB

{m,n}?

{m,n}+ - No backtracking

To match -

[a-z] [-a] [a-]

[^5]

(?>...)

FOL - CNF

```
def getAttr(string):
    expr = '\([^)]+\)'
    matches = re.findall(
    return [m for m in
        if m.alpha()
```

```
def getPred(string):
    expr = '[a-z~]
    return re.find
```

```
def skol(st)
    SC = [f'{chr(C)}'
        range(ord('A')
```

```
matches = re.find
```

```
for match in ma
    st = st.replace(
    for predicate
    attr = getAtt
    if '' .join(a
        st = st.
        SKO
```

```
return skole
```

```
import re
def ftoc(fol):
    st = fol.re
    expr = '\[(
```

Scanned with OKEN Scanner

FOL - CNF

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

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


def skol(st)
    SC = [f'{chr(C)}' for c in range.
          range(ord('A'), ord('Z')+1)]

    matches = re.findall('[∃].',
                         :skolement)

for match in matches[::-1]:
    st = st.replace(match, '')
    for predicate in getP(skolemnum)
        attr = getAttr(pred)
        if ''.join(attr).isupper()
            st = st.replace(match[1],
                 SKOLEM_CONST.pop(0))

    return skolement

import re
def ftoc(fol):
    st = fol.replace("=>", "_")
    expr = '\[(.[^]]+)\]'
```

```
st = re.findall(expr, stmt)

for i, s in enumerate(stmt)
    if '[' in s and ']' not in s:
        stmt[i] += ']'

for s in stmt:
    stmt = stmt.replace(s, fix(s))

while '~' in stmt
    i = stmt.index('~')
    br = stmt.index('[') if '[' in
        stmt else 0
    new_s = st[:br] + new_st if
            br > 0 else new_st
return skolemization(stmt)
```

SAMPLE O/P:

bird (x) ⇒ ~ fly(x)

~bird(x) |~ fly(x)

Satk

12/1/24

## 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(express
                                     -ion)

    self.predicate = predicate
    self.params = params
    self.result = any(self.getConstants())
```

```
def splitExpression(self, Expression)
predicate = getPredicates(expression)[
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 is variable(p) else p for
    p in self.params])})"
    return Fact(f)

class Implication:
```

```
def __init__(self, expr
    self.expression = exp
    l = expression.split(
    self.lhs = [Fact(f) f
        l[0].spl
    self.rhs = Fact(l[1

def evaluate(self, fa
    constants = {}
    new_lhs = []
    for fact in fac
        for val in s
            if val.predi
                fact, p
    for i, v in
        (val.get
        if v
        consta
            fact
    new_lhs.
predicate, attribut
    (self.rhs.expres
    (getAttributes(se

for key in co
    if constants
        attributes

expr = f'{
```

Scanned with OKEN Scanner

```
ion(self, Expression)
edicates(expression)[0]
tributes(expression)[0]
'))', split(',')

, params]

;
result

f):
Variable(c)
self.params]

lf):
iable(v) else
.params]

onstants);
)
s}

pop[0]
else for
)})
```

```
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}'
```

Scanned with OKEN Scanner

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

Sample O/P

Querying criminal(
All facts:
1. american(west)
2. sells(West, M1, None
3. missile(m1)
4. enemy(None, Ameri
5. criminal(West)
6. weapon(M1)
7. owns(None, M1)
8. hostile(None)

Scanned with OKEN Scanner

```
ct[expr)if
lhs) and
tResult() for f in
lse None])


__(self):
cts = set()
nplications = set()

lf, e):
e:
lications.add
cation(e))

.add(Fact(e))
lf.implications:
luate(self.facts)

.add(res)

e):
expression for f
cts])

q!e?:}

icate==Fact(e).
icate:
}.{f}")
```

```
def display(self):
    print("All facts")
    for i, f in enumerate
(set([f.expression for f in self.
    facts])):
        print(f"{i+1}. {f}")
```

Sample O/P

Querying criminal(x):
All facts:
1. american(west)
2. sells(west, M1, None)
3. missile(M1)
4. enemy(None, America)
5. criminal(West)
6. weapon(M1)
7. owns(None, M1)
8. hostile(None)

Scanned with OKEN Scanner

| Sl. No. | Date | Title | Page No. | Teacher's Sign. / Remarks |
|---|---|---|---|---|
| 1. | 22/11/23 | TicTacToe | | |
| 2. | 22/11/23 | Vacuum cleaner. | | |
| 3. | 22/11/23 | BFS 8puzzle | | |
| 4. | 6/12/23. | DLS 8puzzle. | | |
| 6. | 20/12/23 | Propositional logic | | |
| 5. | 13/12/23 | A*, greedy BFS | | |
| 7. | 20/12/23 | Prop. logic | | |
| 8. | 17/1/24 | unification | | |
| 9. | 18/1/24 | FOL-CNF | | |
| 10. | 24/1/24 | Forward chaining in FOL. | | |

S#1 22/11

S#2 6/12

S#3 13/12

S#4 27/12

S#5 17/1/24

Scanned with OKEN Scanner

# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**on**

# ARTIFICIAL INTELLIGENCE
## (22CS5PCAIN)

***Submitted by***

**VAISHNAVI KAMATH(1BM21CS235)**

***in partial fulfillment for the award of the degree of***
**BACHELOR OF ENGINEERING**
***in***
**COMPUTER SCIENCE AND ENGINEERING**

**B.M.S. COLLEGE OF ENGINEERING**
**(Autonomous Institution under VTU)**
**BENGALURU-560019**
**November-2023 to Febrauary-2024**

# B. M. S. College of Engineering,

**Bull Temple Road, Bangalore 560019**

(Affiliated To Visvesvaraya Technological University, Belgaum)

**Department of Computer Science and Engineering**

## CERTIFICATE

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Vaishnavi Kamath (1BM21CS235) during the 5th Semester November-February-2024.

Sandhya A Kulkarni
Assistant Professor
Department of CSE
BMSCE, Bengaluru

# Index Sheet

| | | |
|---|---|---|
| 10 | **Create a knowledgebase consisting of first order logic statements and prove the given query using forward reasoning.** | 40 |

WEEK-2

**Analyse and implement Tic-Tac-Toe 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] + ' | ' + 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():
    global board
    run = True
```

```
    while run:
        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!
```

**Analyse and implement vacuum cleaner agent**

```
def vacuum_world():
    # initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
  goal_state = {'A': '0', 'B': '0'}
  cost = 0


  location_input = input("Enter Location of Vacuum") #user_input of location vacuum is placed
  status_input = input("Enter status of " + location_input) #user_input if location is dirty or clean
  status_input_complement = input("Enter status of other room")
  print("Initial Location Condition" + str(goal_state))


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

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

```
            # suck the dirt and mark it as clean
            goal_state['B'] = '0'
            cost += 1                    #cost for suck
            print("COST for SUCK " + str(cost))
            print("Location B has been Cleaned. ")
        else:
            print("No action" + str(cost))
            # suck and mark clean
            print("Location B is already clean.")

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

else:
    print("Vacuum is placed in location B")
```

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

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

else:
    print(cost)
    # suck and mark clean
    print("Location B is already clean.")

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

```
            print("COST for moving LEFT " + str(cost))
            # suck the dirt and mark it as clean
            goal_state['A'] = '0'
            cost += 1  # cost for suck
            print("Cost for SUCK " + str(cost))
            print("Location A has been Cleaned. ")
        else:
            print("No action " + str(cost))
            # suck and mark clean
            print("Location A is already clean.")

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


vacuum_world()
```

**OUTPUT:**

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

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

## WEEK-3

**Analyse 8 Puzzle problem and implement the same using Breadth First Search Algorithm**

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

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 = 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):
    b = state.index(0)

    d = []

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

    pos_moves_it_can = []

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

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

```
def 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
```

OUTPUT:

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

```
[1, 0, 3, 4, 2, 6, 7, 5, 8]
[1, 2, 3, 4, 0, 6, 7, 5, 8]
[0, 1, 3, 4, 2, 6, 7, 5, 8]
[1, 3, 0, 4, 2, 6, 7, 5, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[1, 2, 3, 0, 4, 6, 7, 5, 8]
[1, 2, 3, 4, 6, 0, 7, 5, 8]
[4, 1, 3, 0, 2, 6, 7, 5, 8]
[1, 3, 6, 4, 2, 0, 7, 5, 8]
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
success
```

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

     [2, 0, 3, 1, 8, 4, 7, 6, 5]
     [2, 8, 3, 1, 0, 4, 7, 6, 5]
     [0, 2, 3, 1, 8, 4, 7, 6, 5]
     [2, 3, 0, 1, 8, 4, 7, 6, 5]
     [2, 8, 3, 1, 6, 4, 7, 0, 5]
     [2, 8, 3, 0, 1, 4, 7, 6, 5]
     [2, 8, 3, 1, 4, 0, 7, 6, 5]
     [1, 2, 3, 0, 8, 4, 7, 6, 5]
     [2, 3, 4, 1, 8, 0, 7, 6, 5]
     [2, 8, 3, 1, 6, 4, 0, 7, 5]
     [2, 8, 3, 1, 6, 4, 7, 5, 0]
     [0, 8, 3, 2, 1, 4, 7, 6, 5]
     [2, 8, 3, 7, 1, 4, 0, 6, 5]
     [2, 8, 0, 1, 4, 3, 7, 6, 5]
     [2, 8, 3, 1, 4, 5, 7, 6, 0]
     [1, 2, 3, 7, 8, 4, 0, 6, 5]
     [1, 2, 3, 8, 0, 4, 7, 6, 5]
     success
```

## WEEK-4

**Analyse Iterative Deepening Search Algorithm. Demonstrate how 8 Puzzle problem could be soloved using this algorithm. Implement the same.**

```
import numpy as np

import pandas as pd


def dfs(src, target, limit, visited_states):
    if src == target:
        return True
    if limit <= 0:
        return False


    visited_states.append(src)
    moves = possible_moves(src, visited_states)


    for move in moves:
        if dfs(move, target, limit - 1, visited_states):
```

```
            return True

    return False

def possible_moves(state, visited_states):
    b = state.index(-1)
    d = []

    if b not in [0, 1, 2]:
        d += 'u'
    if b not in [6, 7, 8]:
        d += 'd'
    if b not in [2, 5, 8]:
        d += 'r'
    if b not in [0, 3, 6]:
        d += 'l'

    pos_moves = []

    for move in d:
        pos_moves.append(gen(state, move, b))

    return [move for move in pos_moves if move not in visited_states]

def gen(state, move, blank):
    temp = state.copy()

    if move == 'u':
        temp[blank - 3], temp[blank] = temp[blank], temp[blank - 3]
    elif move == 'd':
```

```
        temp[blank + 3], temp[blank] = temp[blank], temp[blank + 3]
    elif move == 'r':
        temp[blank + 1], temp[blank] = temp[blank], temp[blank + 1]
    elif move == 'l':
        temp[blank - 1], temp[blank] = temp[blank], temp[blank - 1]

    return temp

def iddfs(src, target, depth):
    for i in range(depth):
        visited_states = []
        if dfs(src, target, i + 1, visited_states):
            return True
    return False
```

OUTPUT:

```
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
```

```
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
```

```
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)
```

```
True
```

WEEK-5

**Analyse best first search and A* algorithm. Implement 8 puzzle problem using both algorithms.**

```
import heapq


class Node:
    def __init__(self, state, level, heuristic):
        self.state = state
        self.level = level
        self.heuristic = heuristic


    def __lt__(self, other):
```

```
        return self.heuristic < other.heuristic

def generate_child(node):
    x, y = find_blank(node.state)
    moves = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]
    children = []

    for move in moves:
        child_state = move_blank(node.state, (x, y), move)
        if child_state is not None:
            h = calculate_heuristic(child_state)
            child_node = Node(child_state, node.level + 1, h)
            children.append(child_node)

    return children

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def move_blank(state, src, dest):
    x1, y1 = src
    x2, y2 = dest

    if 0 <= x2 < 3 and 0 <= y2 < 3:
        new_state = [row[:] for row in state]
        new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
        return new_state
```

```
    else:
        return None

def calculate_heuristic(state):
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    h = 0

    for i in range(3):
        for j in range(3):
            if state[i][j] != goal_state[i][j] and state[i][j] != 0:
                h += 1

    return h

def best_first_search(initial_state):
    start_node = Node(initial_state, 0, calculate_heuristic(initial_state))
    open_list = [start_node]
    closed_set = set()

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == [[1, 2, 3], [4, 5, 6], [7, 8, 0]]:
            return current_node

        closed_set.add(tuple(map(tuple, current_node.state)))

        for child in generate_child(current_node):
            if tuple(map(tuple, child.state)) not in closed_set:
                heapq.heappush(open_list, child)
```

```
    return None

initial_state = [[1, 2, 3], [0, 4, 6], [7, 5, 8]]
solution_node = best_first_search(initial_state)

if solution_node:
    print("Solution found in", solution_node.level, "moves.")
    print("Path:")
    for row in solution_node.state:
        print(row)
else:
    print("No solution found.")
```

OUTPUT:

```
Solution found in 3 moves.
Path:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

**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 0 <= x2 < len(self.data) and 0 <= 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("")
        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)
```

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 _
```

## WEEK-6

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

def evaluate_first(premise, conclusion):

```
    # Create all possible models for the variables p, q, and r
    models = [
        {'p': False, 'q': False, 'r': False},
        {'p': False, 'q': False, 'r': True},
        {'p': False, 'q': True, 'r': False},
        {'p': False, 'q': True, 'r': True},
        {'p': True, 'q': False, 'r': False},
        {'p': True, 'q': False, 'r': True},
        {'p': True, 'q': True, 'r': False},
        {'p': True, 'q': True, 'r': True}
    ]

    # Check if the premise logically entails the conclusion
    entails = True  # Initially assume the premise entails the conclusion
    for model in models:
        if evaluate_expression(premise, model) != evaluate_expression(conclusion, model):
            entails = False  # If any model disagrees, premise does not entail conclusion
            break

    # Return the result
    return entails

# Define a function to evaluate logical expressions for the second input
def evaluate_second(premise, conclusion):
    # Generate all possible truth assignments to p, q, and r
    models = [
        {'p': p, 'q': q, 'r': r}
        for p in [True, False]
        for q in [True, False]
        for r in [True, False]
```

```
    ]

    # Check if the conclusion holds in every model where the premise is true
    entails = all(evaluate_expression(premise, model) for model in models if
evaluate_expression(conclusion, model) and evaluate_expression(premise, model))

    # Return the result
    return entails

# Define a function to evaluate logical expressions based on the given expression and model
def evaluate_expression(expression, model):
    # Evaluate the logical expression recursively using the given model
    if isinstance(expression, str):
        # Base case: if it's a single variable or literal
        return model.get(expression)
    elif isinstance(expression, tuple):
        op = expression[0]  # Get the operator
        if op == 'not':
            return not evaluate_expression(expression[1], model)  # Negation
        elif op == 'and':
            return evaluate_expression(expression[1], model) and
evaluate_expression(expression[2], model)  # Conjunction
        elif op == 'if':
            return (not evaluate_expression(expression[1], model)) or
evaluate_expression(expression[2], model)  # Implication
        elif op == 'or':
            return evaluate_expression(expression[1], model) or
evaluate_expression(expression[2], model)  # Disjunction (OR)

# Premise and conclusion for the first and second inputs
first_premise = ('and', ('or', 'p', 'q'), ('or', ('not', 'r'), 'p'))  # (p OR q) AND (NOT r OR p)
first_conclusion = ('and', 'p', 'r')  # p AND r
```

```
second_premise = ('and', ('or', ('not', 'q'), ('not', 'p'), 'r'), ('and', ('not', 'q'), 'p'), 'q')
second_conclusion = 'r'

# Evaluate both inputs using the merged function
result_first = evaluate_first(first_premise, first_conclusion)
result_second = evaluate_second(second_premise, second_conclusion)

# Print the results for both inputs
if result_first:
    print("For the first input: The knowledge base entails the query.")
else:
    print("For the first input: The knowledge base does not entail the query.")

if result_second:
    print("For the second input: The knowledge base entails the query.")
else:
    print("For the second input: The knowledge base does not entail the query.")
```

OUTPUT:

```
>>>
============ RESTART: C:/Users/Admin/Desktop/1BM21CS205 AI LAB/p5.py ===========
For the first input: The knowledge base does not entail the query.
For the second input: The knowledge base entails the query.
>>>
```

## WEEK-7

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

```
import re
```

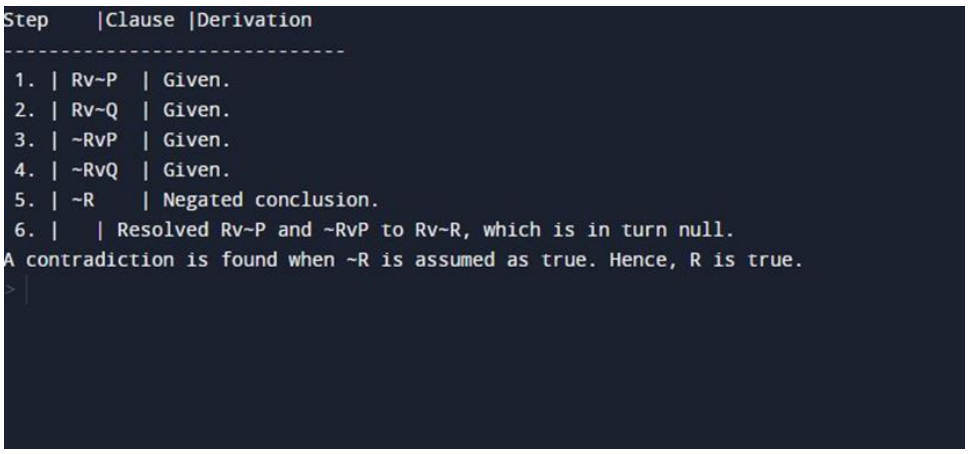
```
def main(rules, goal):
    rules = rules.split(' ')
    steps = resolve(rules, goal)
    print('\nStep\t|Clause\t|Derivation\t')
    print('-' * 30)
    i = 1
    for step in steps:
        print(f' {i}.\t| {step}\t| {steps[step]}\t')
        i += 1
def negate(term):
    return f'~{term}' if term[0] != '~' 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
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
```

OUTPUT

```
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')
```

```
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.
```

**Implement unification in first order logic.**

```
import re

def getAttributes(expression):
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression[:-1]
    expression = re.split("(?<!\(.),(?!.\))", expression)
    return expression

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)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    predicate = getInitialPredicate(exp)
    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:
            return False

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

    if isConstant(exp2):
```

```
    return [(exp2, exp1)]

if isVariable(exp1):
    if checkOccurs(exp1, exp2):
        return False
    else:
        return [(exp2, exp1)]

if isVariable(exp2):
    if checkOccurs(exp2, exp1):
        return False
    else:
        return [(exp1, exp2)]

if getInitialPredicate(exp1) != getInitialPredicate(exp2):
    print("Predicates do not match. Cannot be unified")
    return False

attributeCount1 = len(getAttributes(exp1))
attributeCount2 = len(getAttributes(exp2))
if attributeCount1 != attributeCount2:
    return False

head1 = getFirstPart(exp1)
head2 = getFirstPart(exp2)
initialSubstitution = unify(head1, head2)
if not initialSubstitution:
    return False
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 False

    initialSubstitution.extend(remainingSubstitution)
    return initialSubstitution
```

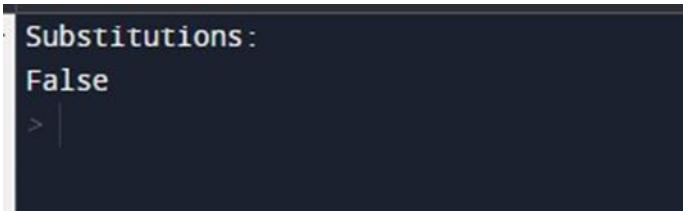
OUTPUT

exp1 = "knows(A,x)"

exp2= "knows(y,Y)"


exp1 = "like(A,y)"

exp2 = "like(K,g(x))"



## WEEK-9

**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 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:
            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
    return Skolemization(statement)
```

OUTPUT:

## WEEK-10

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

```
import re

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

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

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

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

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

    def getResult(self):
        return self.result

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

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

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

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

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

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

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

    def query(self, e):
```

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

    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

Shell

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