**B.M.S COLLEGE OF ENGINEERING**

**BENGALURU** Autonomous Institute, Affiliated to VTU



Lab Record

**Artificial Intelligence**

**(22CS5PCAIN)**

*Submitted in partial fulfillment for the 5th Semester Laboratory*

Bachelor of Technology

in

Computer Science and Engineering

*Submitted by:*

**Akram**

1BM21CS013

Department of Computer Science and Engineering . B.M.S College of Engineering

Bull Temple Road, Basavanagudi, Bangalore 560 019

Nov 2023-Feb 2024

**B.M.S COLLEGE OF ENGINEERING**

**DEPARTMENT OF COMPUTER SCIENCE** **AND ENGINEERING**



***CERTIFICATE***

This is to certify that the Artificial Intelligence (22CS5PCAIN) laboratory has been carried out by Akram (1BM21CS013) during the 5th Semester September-January 2021.

Signature of the Faculty Incharge:

Gauri Kalnoor

Assistant Professor

Department of Computer Science and Engineering

B.M.S. College of Engineering, Bangalore

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Sl. No.** | **Title** | **Page No.** |
| 1. | Tic Tac Toe | 1 |
| 2. | 8 Puzzle Breadth First Search Algorithm | 3 |
| 3. | 8 Puzzle Iterative Deepening Search Algorithm | 5 |
| 4. | 8 Puzzle A\* Search Algorithm | 9 |
| 5. | Vacuum Cleaner | 12 |
| 6. | Knowledge Base Entailment | 13 |
| 7. | Knowledge Base Resolution | 16 |
| 8. | Unification | 17 |
| 9. | FOL to CNF | 18 |
| 10. | Forward reasoning | 20 |

**1.TIC-TAC-TOE**

def printBoard(board):

print(board[1] + '|' + board[2] + '|' + board[3])

print('-+-+-')

print(board[4] + '|' + board[5] + '|' + board[6])

print('-+-+-')

print(board[7] + '|' + board[8] + '|' + board[9])

print("\n")

def spaceIsFree(position):

if board[position] == ' ':

return True

else:

return False

def insertLetter(letter, position):

if spaceIsFree(position):

board[position] = letter

printBoard(board)

if (checkDraw()):

print("Draw!")

exit()

if checkForWin():

if letter == 'X':

print("Bot wins!")

exit()

else:

print("Player wins!")

exit()

return

else:

print("Can't insert there!")

position = int(input("Please enter new position: "))

insertLetter(letter, position)

return

def checkForWin():

if (board[1] == board[2] and board[1] == board[3] and board[1] != ' '):

return True

elif (board[4] == board[5] and board[4] == board[6] and board[4] != ' '):

return True

elif (board[7] == board[8] and board[7] == board[9] and board[7] != ' '):

return True

elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):

return True

elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):

return True

elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):

return True

elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):

return True

elif (board[7] == board[5] and board[7] == board[3] and board[7] != ' '):

return True

else:

return False

def checkWhichMarkWon(mark):

if board[1] == board[2] and board[1] == board[3] and board[1] == mark:

return True

elif (board[4] == board[5] and board[4] == board[6] and board[4] == mark):

return True

elif (board[7] == board[8] and board[7] == board[9] and board[7] == mark):

return True

elif (board[1] == board[4] and board[1] == board[7] and board[1] == mark):

return True

elif (board[2] == board[5] and board[2] == board[8] and board[2] == mark):

return True

elif (board[3] == board[6] and board[3] == board[9] and board[3] == mark):

return True

elif (board[1] == board[5] and board[1] == board[9] and board[1] == mark):

return True

elif (board[7] == board[5] and board[7] == board[3] and board[7] == mark):

return True

else:

return False

def checkDraw():

for key in board.keys():

if (board[key] == ' '):

return False

return True

def playerMove():

position = int(input("Enter the position for 'O': "))

insertLetter(player, position)

return

def compMove():

bestScore = -800

bestMove = 0

for key in board.keys():

if (board[key] == ' '):

board[key] = bot

score = minimax(board, 0, False)

board[key] = ' '

if (score > bestScore):

bestScore = score

bestMove = key

insertLetter(bot, bestMove)

return

def minimax(board, depth, isMaximizing):

if (checkWhichMarkWon(bot)):

return 1

elif (checkWhichMarkWon(player)):

return -1

elif (checkDraw()):

return 0

if (isMaximizing):

bestScore = -800

for key in board.keys():

if (board[key] == ' '):

board[key] = bot

score = minimax(board, depth + 1, False)

board[key] = ' '

if (score > bestScore):

bestScore = score

return bestScore

else:

bestScore = 800

for key in board.keys():

if (board[key] == ' '):

board[key] = player

score = minimax(board, depth + 1, True)

board[key] = ' '

if (score < bestScore):

bestScore = score

return bestScore

board = {1: ' ', 2: ' ', 3: ' ',

4: ' ', 5: ' ', 6: ' ',

7: ' ', 8: ' ', 9: ' '}

printBoard(board)

print("Computer goes first! Good luck.")

print("Positions are as follow:")

print("1, 2, 3 ")

print("4, 5, 6 ")

print("7, 8, 9 ")

print("\n")

player = 'O'

bot = 'X'

global firstComputerMove

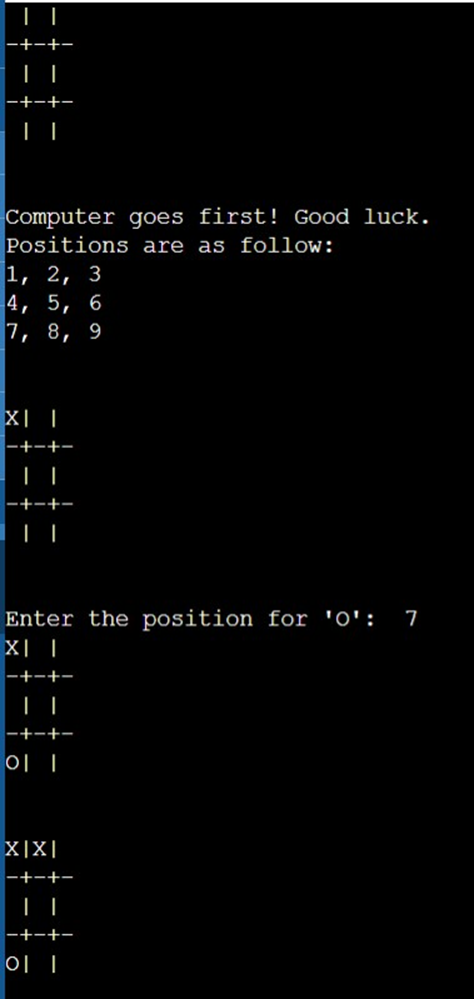
firstComputerMove = True

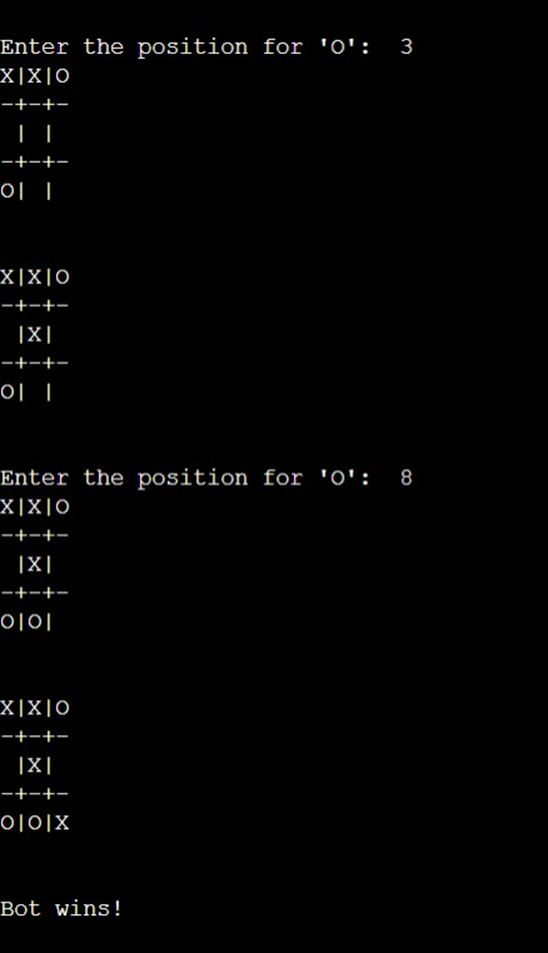
while not checkForWin():

compMove()

playerMove()

OUTPUT





**2. 8 Puzzle**  
**(bfs)**

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 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 [-1,1,2]:

d.append('u')

if b not in [6,7,8]:

d.append('d')

if b not in [-1,3,6]:

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 move\_it\_can in pos\_moves\_it\_can if move\_it\_can not in visited\_states]

def gen(state, m, b):

temp = state.copy()

if m=='d':

temp[b+3],temp[b] = temp[b],temp[b+3]

if m=='u':

temp[b-3],temp[b] = temp[b],temp[b-3]

if m=='l':

temp[b-1],temp[b] = temp[b],temp[b-1]

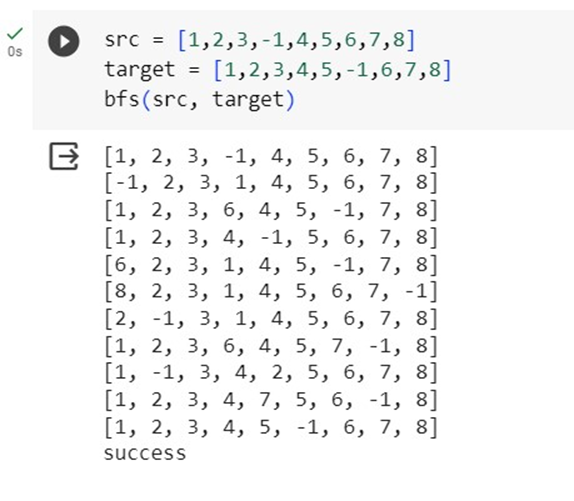
if m=='r':

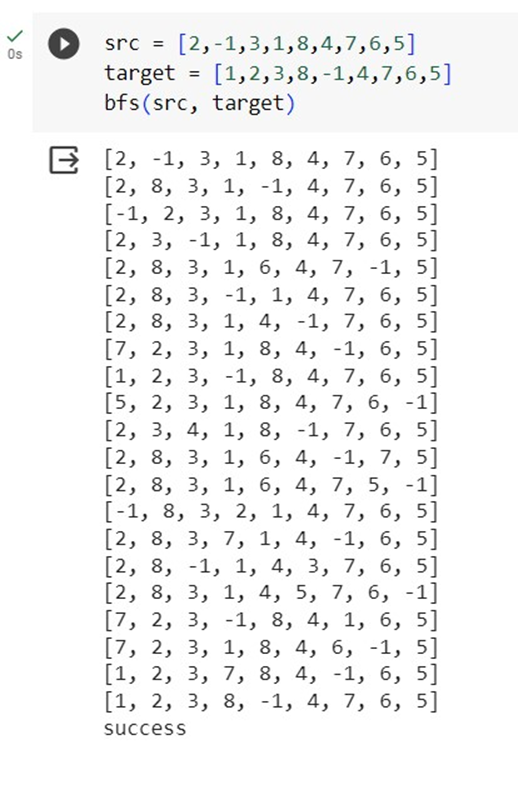
temp[b+1],temp[b] = temp[b],temp[b+1]

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

return temp

OUTPUT





**3. Implement Iterative deepening search algorithm**

def dfs(src,target,limit,visited\_states):

if src == target:

return True

if limit <= 0:

return False

visited\_states.append(src)

moves = possible\_moves(src,visited\_states)

for move in moves:

if dfs(move, target, limit-1, visited\_states):

return True

return False

def possible\_moves(state,visited\_states):

b = state.index(-1)

d = []

if b not in [0,1,2]:

d += 'u'

if b not in [6,7,8]:

d += 'd'

if b not in [2,5,8]:

d += 'r'

if b not in [0,3,6]:

d += 'l'

pos\_moves = []

for move in d:

pos\_moves.append(gen(state,move,b))

return [move for move in pos\_moves if move not in visited\_states]

def gen(state, move, blank):

temp = state.copy()

if move == 'u':

temp[blank-3], temp[blank] = temp[blank], temp[blank-3]

if move == 'd':

temp[blank+3], temp[blank] = temp[blank], temp[blank+3]

if move == 'r':

temp[blank+1], temp[blank] = temp[blank], temp[blank+1]

if move == 'l':

temp[blank-1], temp[blank] = temp[blank], temp[blank-1]

return temp

def iddfs(src,target,depth):

for i in range(depth):

visited\_states = []

if dfs(src,target,i+1,visited\_states):

return True

return False

src = []

target=[]

n = int(input("Enter number of elements : "))

print("Enter source elements")

for i in range(0, n):

ele = int(input())

src.append(ele)

print("Enter target elements")

for i in range(0, n):

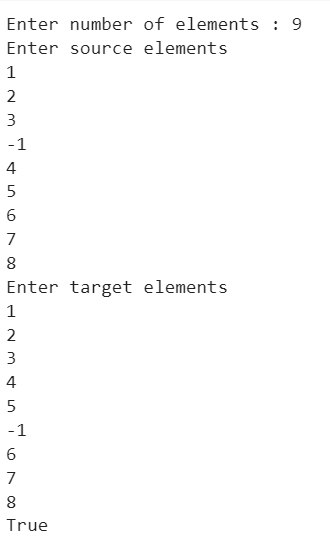
ele = int(input())

target.append(ele)

depth=8

iddfs(src, target,depth)

OUTPUT



**4. 8 Puzzle A\* Search Algorithm**

class Node:

def \_\_init\_\_(self, data, level, fval):

# Initialize the node with the data ,level of the node and the calculated fvalue

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

# Generate hild nodes from the given node by moving the blank space

# either in the four direction {up,down,left,right}

x, y = self.find(self.data, '\_')

# val\_list contains position values for moving the blank space in either of

# the 4 direction [up,down,left,right] respectively.

val\_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]

children = []

for i in val\_list:

child = self.shuffle(self.data, x, y, i[0], i[1])

if child is not None:

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

children.append(child\_node)

return children

def shuffle(self, puz, x1, y1, x2, y2):

# Move the blank space in the given direction and if the position value are out

# of limits the return None

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp\_puz = []

temp\_puz = self.copy(puz)

temp = temp\_puz[x2][y2]

temp\_puz[x2][y2] = temp\_puz[x1][y1]

temp\_puz[x1][y1] = temp

return temp\_puz

else:

return None

def copy(self, root):

# copy function to create a similar matrix of the given node

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self, puz, x):

# Specifically used to find the position of the blank space

for i in range(0, len(self.data)):

for j in range(0, len(self.data)):

if puz[i][j] == x:

return i, j

class Puzzle:

def \_\_init\_\_(self, size):

# Initialize the puzzle size by the the specified size,open and closed lists to empty

self.n = size

self.open = []

self.closed = []

def accept(self):

# Accepts the puzzle from the user

puz = []

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

temp = input().split(" ")

puz.append(temp)

return puz

def f(self, start, goal):

# Heuristic function to calculate Heuristic value f(x) = h(x) + g(x)

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

def h(self, start, goal):

# Calculates the difference between the given puzzles

temp = 0

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

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

if start[i][j] != goal[i][j] and start[i][j] != '\_':

temp += 1

return temp

def process(self):

# Accept Start and Goal Puzzle state

print("enter the start state matrix \n")

start = self.accept()

print("enter the goal state matrix \n")

goal = self.accept()

start = Node(start, 0, 0)

start.fval = self.f(start, goal)

# put the start node in the open list

self.open.append(start)

print("\n\n")

while True:

cur = self.open[0]

print("==================================================\n")

for i in cur.data:

for j in i:

print(j, end=" ")

print("")

# if the difference between current and goal node is 0 we have reached the goal node

if (self.h(cur.data, goal) == 0):

break

for i in cur.generate\_child():

i.fval = self.f(i, goal)

self.open.append(i)

self.closed.append(cur)

del self.open[0]

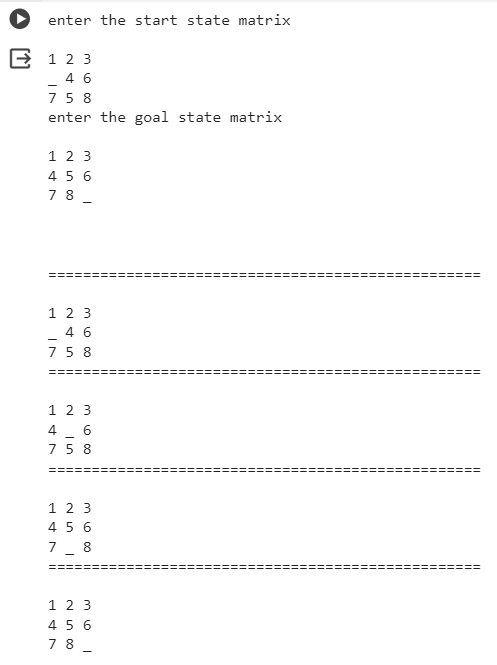
# sort the open list based on f value

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

puz = Puzzle(3)

puz.process()

OUTPUT



**5.Vaccum cleaner**

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

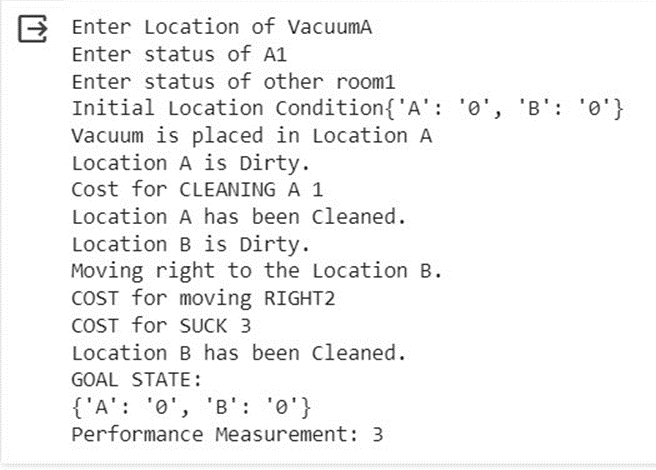
print("GOAL STATE: ")

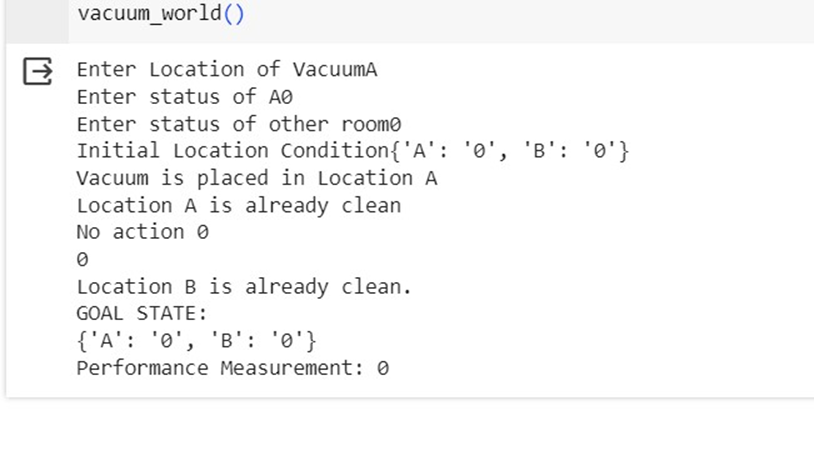
print(goal\_state)

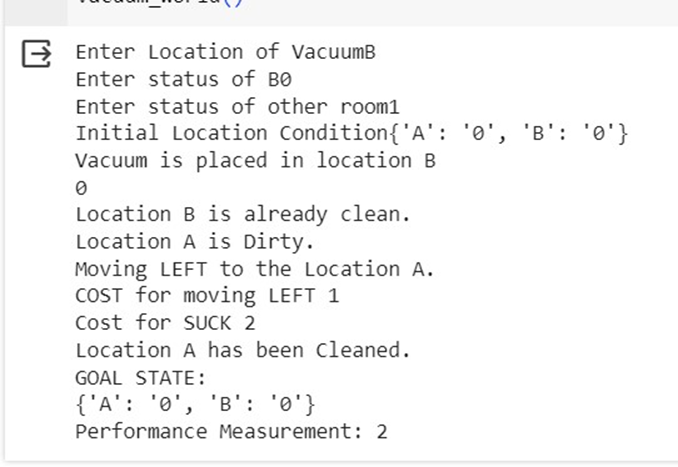
print("Performance Measurement: " + str(cost))

vacuum\_world()

OUTPUT







**6. Knowledge Base Entailment**

def tell(kb, rule):

kb.append(rule)

combinations = [(True, True, True), (True, True, False),

(True, False, True), (True, False, False),

(False, True, True), (False, True, False),

(False, False, True), (False, False, False)]

def ask(kb, q):

for c in combinations:

s = all(rule(c) for rule in kb)

f = q(c)

print(s, f)

if s != f and s != False:

return 'Does not entail'

return 'Entails'

kb = []

# Get user input for Rule 1

rule\_str = input("Enter Rule 1 as a lambda function (e.g., lambda x: x[0] or x[1] and (x[0] and x[1]): ")

r1 = eval(rule\_str)

tell(kb, r1)

# Get user input for Rule 2

#rule\_str = input("Enter Rule 2 as a lambda function (e.g., lambda x: (x[0] or x[1]) and x[2]): ")

#r2 = eval(rule\_str)

#tell(kb, r2)

# Get user input for Query

query\_str = input("Enter Query as a lambda function (e.g., lambda x: x[0] and x[1] and (x[0] or x[1]): ")

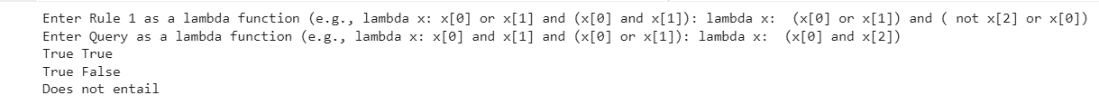
q = eval(query\_str)

# Ask KB Query

result = ask(kb, q)

print(result)

OUTPUT



**7. Knowledge Base 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

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

goal = 'R'

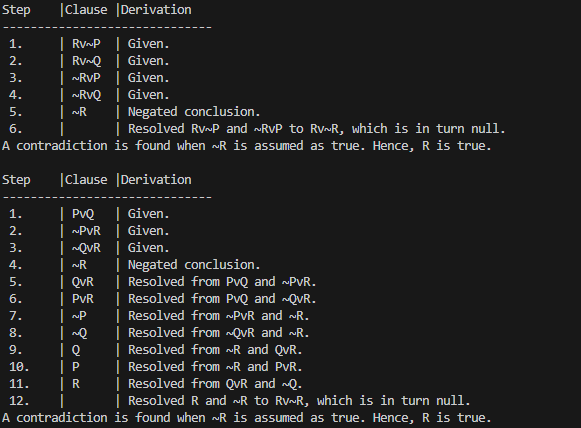
main(rules, goal)

rules = 'PvQ ~PvR ~QvR' #P=vQ, P=>Q : ~PvQ, Q=>R, ~QvR

goal = 'R'

main(rules, goal)

OUTPUT



**8. Unification**

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

exp1 = "knows(A,x)"

exp2 = "knows(y,mother(y))"

substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)

OUTPUT



**9. FOL to CNF**

import re

def getAttributes(string):

expr = '\([^)]+\)'

matches = re.findall(expr, string)

return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):

expr = '[a-z~]+\([A-Za-z,]+\)'

return re.findall(expr, string)

def 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

def fol\_to\_cnf(fol):

statement = fol.replace("=>", "-")

expr = '\[([^]]+)\]'

statements = re.findall(expr, statement)

print(statements)

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)

print(fol\_to\_cnf("bird(x)=>~fly(x)"))

print(fol\_to\_cnf("∃x[bird(x)=>~fly(x)]"))

OUTPUT



**10. 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}')

kb\_ = KB()

kb\_.tell('king(x)&greedy(x)=>evil(x)')

kb\_.tell('king(John)')

kb\_.tell('greedy(John)')

kb\_.tell('king(Richard)')

kb\_.query('evil(x)')

OUTPUT

