**Artificial Intelligence Lab Report**

****

***Submitted by***

**MD SURAJ KUMAR(1BM20CS079)**

**Batch: B2**

**Course: Artificial Intelligence**

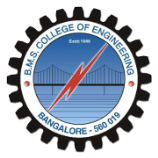
**Course Code: 20CS5PCAIP**

**Sem & Section: 5B**

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

****

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

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**2022-2023**

**Table of contents**

| **Program Number** | **Program Title** | **Page Number** |
| --- | --- | --- |
| **1** | **Tic-Tac\_Toe** | **3-7** |
| **2** | **8-Puzzle BFS** | **8-11** |
| **3** | **8-Puzzle IDDFS** | **12-15** |
| **4** | **8-Puzzle A\*** | **16-19** |
| **5** | **Vacuum Cleaner** | **20-22** |
| **6** | **Knowledge Base-entailment** | **23-25** |
| **7** | **Knowledge Base - Resolution** | **26-35** |
| **8** | **Unification in First Order Logic** | **36-38** |
| **9** | **First Order Logic to Conjunctive Normal Form** | **39-41** |
| **10** | **Forward Reasoning** | **42-45** |

**LAB PROGRAM 1**

**Implement Tic –Tac –Toe Game.**

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

**Code:**

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'

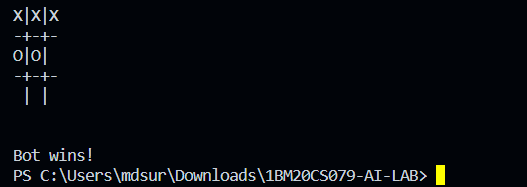
global firstComputerMove

firstComputerMove = True

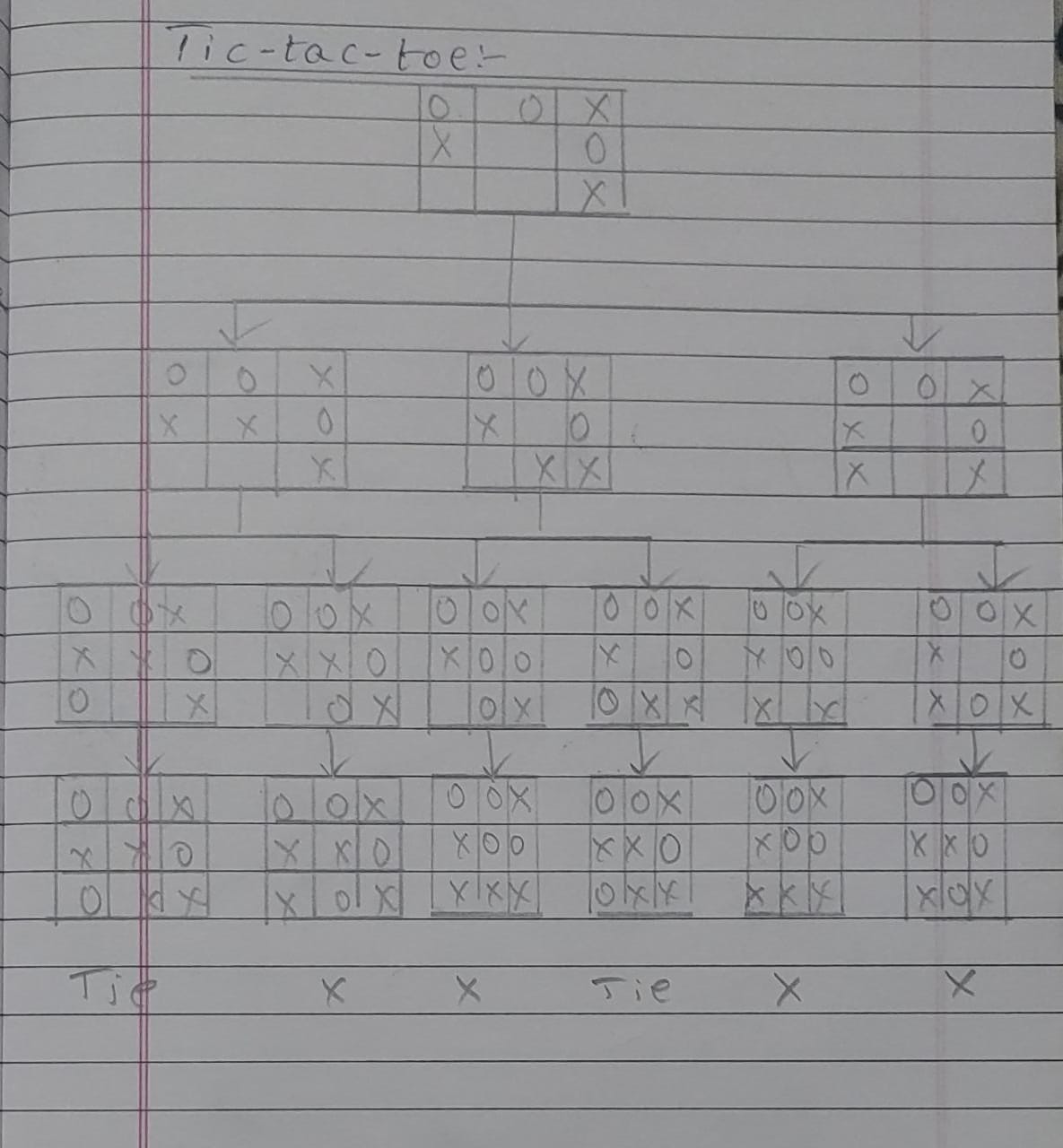
while not checkForWin():

compMove()

playerMove()

**Output Snapshot**

****

**State Space Tree**

**LAB PROGRAM 2**

**Solve 8 puzzle problem.**

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

**Code:**

from queue import Queue

class Puzzle:

goal\_state=[1,2,3,5,8,6,0,7,4]

heuristic=None

evaluation\_function=None

num\_of\_instances=0

def \_\_init\_\_(self,state,parent,action,path\_cost):

self.parent=parent

self.state=state

self.action=action

if parent:

self.path\_cost = parent.path\_cost + path\_cost

else:

self.path\_cost = path\_cost

Puzzle.num\_of\_instances+=1

def \_\_str\_\_(self):

return str(self.state[0:3])+'\n'+str(self.state[3:6])+'\n'+str(self.state[6:9])

def goal\_test(self):

if self.state == self.goal\_state:

return True

return False

@staticmethod

def find\_legal\_actions(i,j):

legal\_action = ['U', 'D', 'L', 'R']

if i == 0: # up is disable

legal\_action.remove('U')

elif i == 2: # down is disable

legal\_action.remove('D')

if j == 0:

legal\_action.remove('L')

elif j == 2:

legal\_action.remove('R')

return legal\_action

def generate\_child(self):

children=[]

x = self.state.index(0)

i = int(x / 3)

j = int(x % 3)

legal\_actions=self.find\_legal\_actions(i,j)

for action in legal\_actions:

new\_state = self.state.copy()

if action == 'U':

new\_state[x], new\_state[x-3] = new\_state[x-3], new\_state[x]

elif action == 'D':

new\_state[x], new\_state[x+3] = new\_state[x+3], new\_state[x]

elif action == 'L':

new\_state[x], new\_state[x-1] = new\_state[x-1], new\_state[x]

elif action == 'R':

new\_state[x], new\_state[x+1] = new\_state[x+1], new\_state[x]

children.append(Puzzle(new\_state,self,action,1))

return children

def find\_solution(self):

solution = []

solution.append(self.action)

path = self

while path.parent != None:

path = path.parent

solution.append(path.action)

solution = solution[:-1]

solution.reverse()

return solution

def breadth\_first\_search(initial\_state):

start\_node = Puzzle(initial\_state, None, None, 0)

if start\_node.goal\_test():

return start\_node.find\_solution()

q = Queue()

q.put(start\_node)

explored=[]

while not(q.empty()):

node=q.get()

explored.append(node.state)

children=node.generate\_child()

for child in children:

if child.state not in explored:

node.\_\_str\_\_()

if child.goal\_test():

return child.find\_solution()

q.put(child)

return

state = [1, 2, 3,

5, 6, 0,

7, 8, 4]

Puzzle.num\_of\_instances=0

bfs=breadth\_first\_search(state)

print('BFS:', bfs)

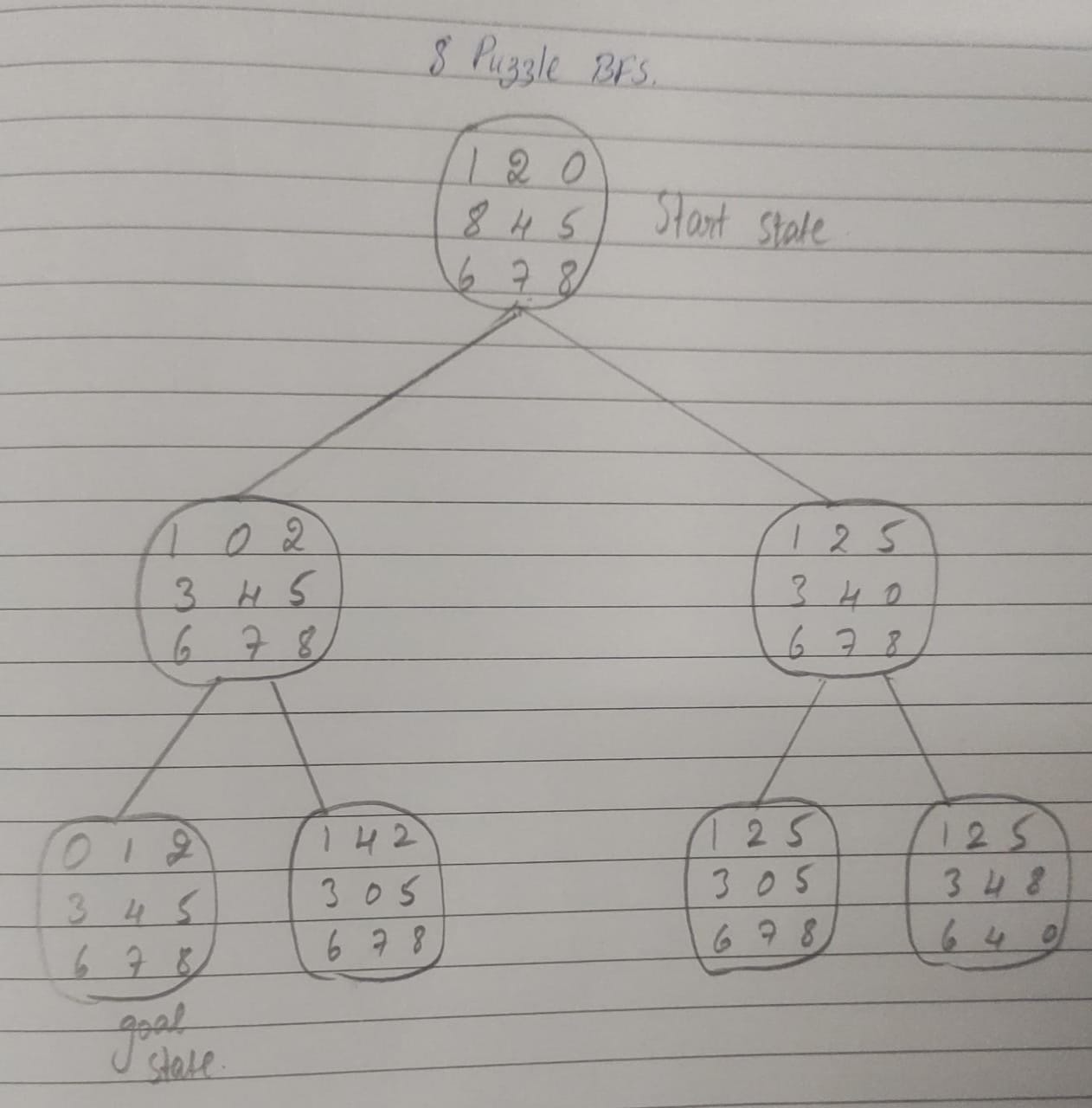
print('space:',Puzzle.num\_of\_instances)

print()

**Output Snapshot**

****

**State Space Tree**

****

**LAB PROGRAM 3**

**Implement Iterative deepening search algorithm.**

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

**Code:**

import itertools

import random

import time

def id\_dfs(puzzle, goal, get\_moves):

def idfs(path, depth):

if depth == 0:

return

if path[-1] == goal:

return path

for move in get\_moves(path[-1]):

if move not in path:

next\_path = idfs(path + [move], depth - 1)

if next\_path:

#print(next\_path, end="")

return next\_path

for depth in itertools.count():

path = idfs([puzzle], depth)

if path:

#print(path)

return path

def num\_matrix(rows, cols, steps=25):

nums = list(range(1, rows \* cols)) + [0]

goal = [ nums[i:i+rows] for i in range(0, len(nums), rows) ]

get\_moves = num\_moves(rows, cols)

puzzle = goal

for steps in range(steps):

puzzle = random.choice(get\_moves(puzzle))

return puzzle, goal

def num\_moves(rows, cols):

def get\_moves(subject):

moves = []

zrow, zcol = next((r, c)

for r, l in enumerate(subject)

for c, v in enumerate(l) if v == 0)

def swap(row, col):

import copy

s = copy.deepcopy(subject)

s[zrow][zcol], s[row][col] = s[row][col], s[zrow][zcol]

return s

if zrow > 0:

moves.append(swap(zrow - 1, zcol))

if zcol < cols - 1:

moves.append(swap(zrow, zcol + 1))

if zrow < rows - 1:

moves.append(swap(zrow + 1, zcol))

if zcol > 0:

moves.append(swap(zrow, zcol - 1))

return moves

return get\_moves

if \_\_name\_\_ == '\_\_main\_\_':

reps = 25

total\_time = 0

for i in range(reps):

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

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

puzzle,goal = num\_matrix(3,3)

t0 = time.time()

solution = id\_dfs(puzzle, goal, num\_moves(3, 3))

t1 = time.time()

total\_time += t1 - t0

total\_time /= reps

print("Goal State: ")

for i in goal:

print(i, end="\n")

print("Starting State: ")

for i in puzzle:

print(i, end="\n")

print("Solution: ")

for i in solution:

print("")

print(" | ")

print(" | ")

print(" \\\'/ \n")

for j in i:

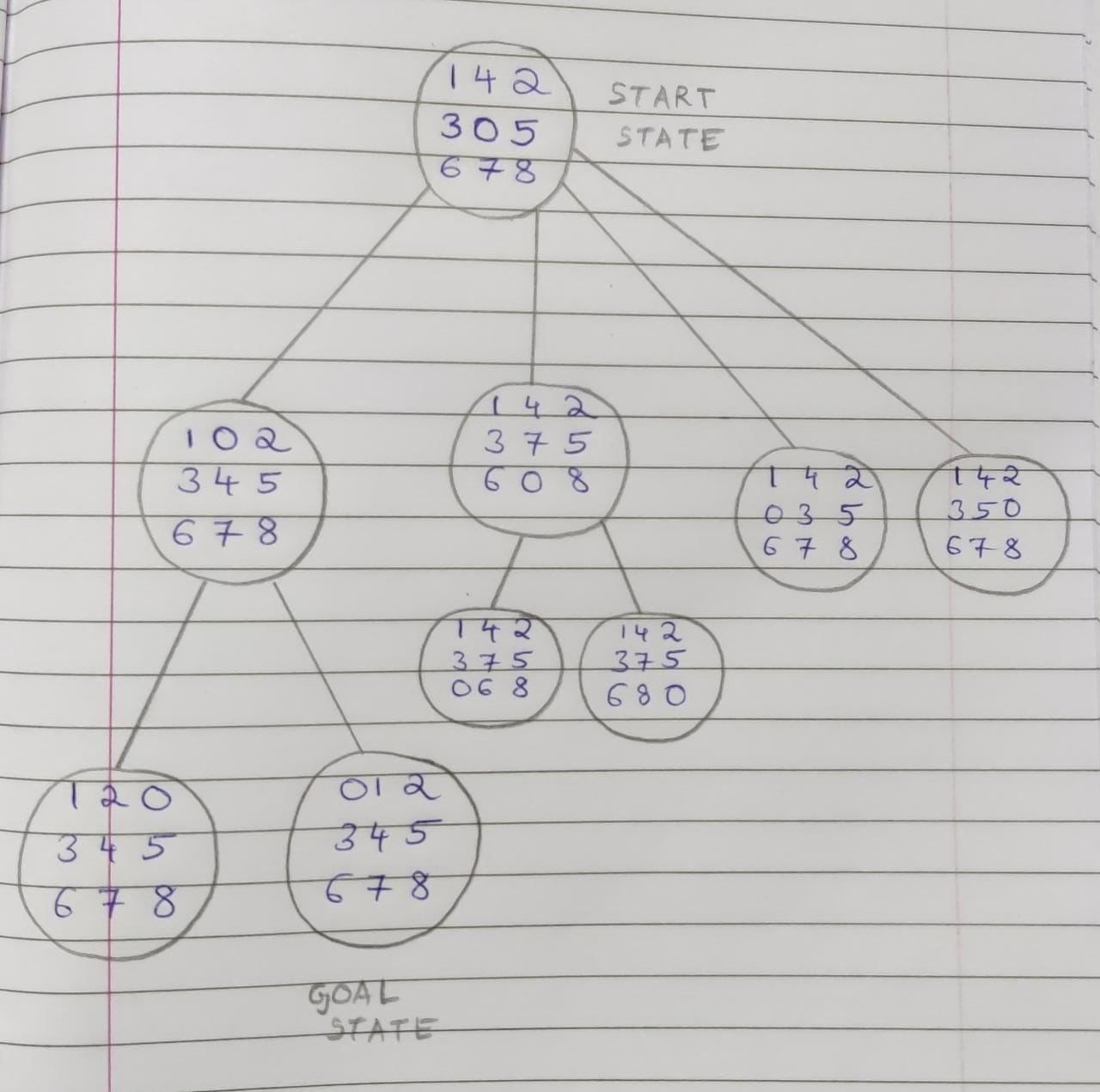
print(j)

print('Puzzle solved using iterative depth first search in', total\_time, 'seconds.') # 0.20 seconds

**Output Snapshot**

****

**State Space Tree**

****

**LAB PROGRAM 4**

**Implement A\* search algorithm.**

**Objective:** The a\* algorithm takes into account both the cost to go to goal from present state as well the cost already taken to reach the present state. In 8 puzzle problem, both depth and number of misplaced tiles are considered to take decision about the next state that has to be visited.

**Code:**

class Node:

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

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

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

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

children = []

for i in val\_list:

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

if child is not None:

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

children.append(child\_node)

return children

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

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

temp\_puz = []

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 accept(self):

puz = []

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

temp = input().split(" ")

puz.append(temp)

return puz

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):

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)

self.open.append(start)

while True:

cur = self.open[0]

print("")

print(" | ")

print(" | ")

print(" \\\'/ \n")

for i in cur.data:

for j in i:

print(j,end=" ")

print("")

if(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 opne list based on f value """

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

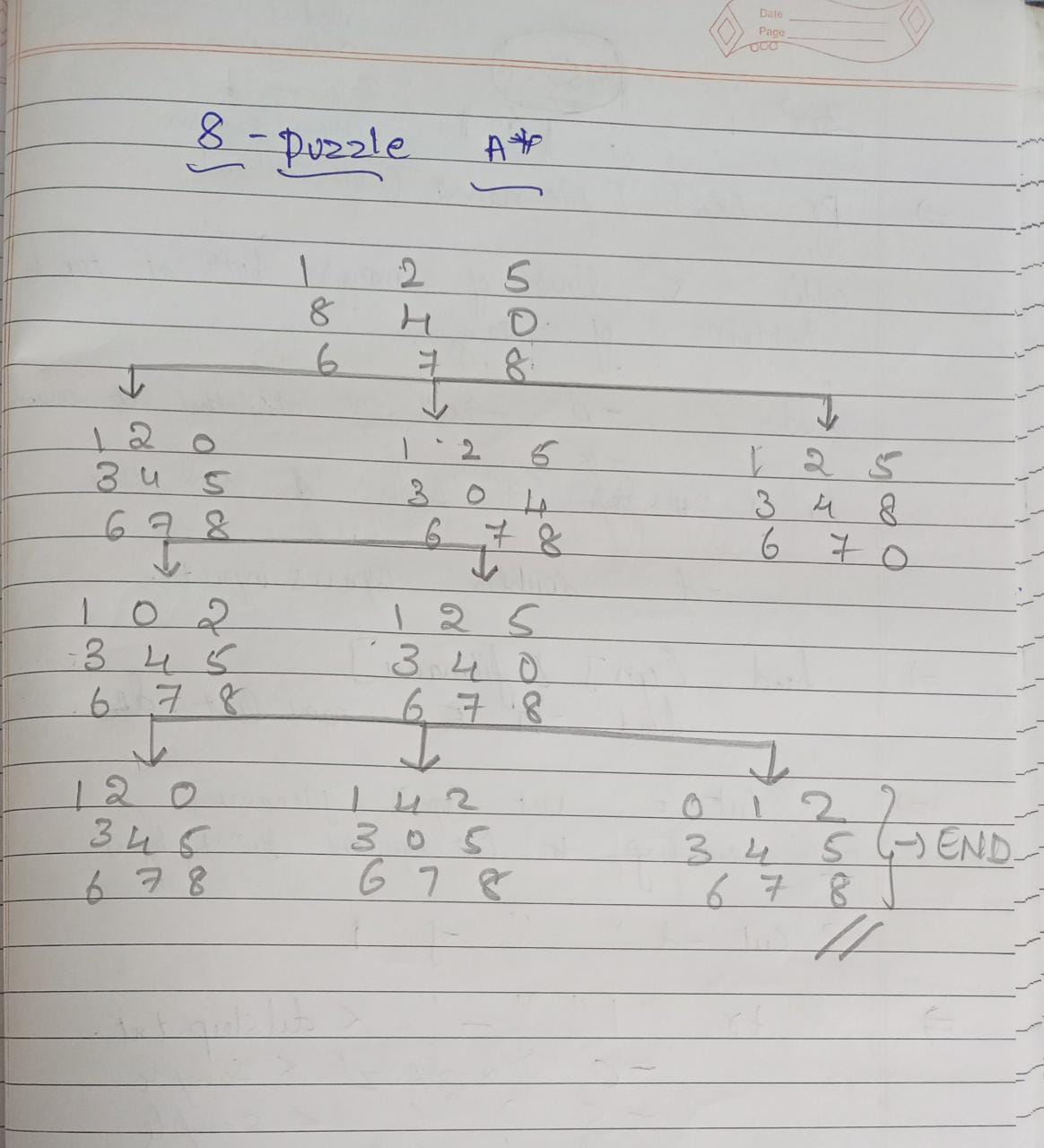
puz = Puzzle(3)

puz.process()

**Output Snapshot**

****

**State Space Tree**

****

**LAB PROGRAM 5**

**Implement vacuum cleaner agent.**

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

**Code:**

def vacuum\_world():

goal\_state = {'A': '0', 'B': '0'}

cost = 0

actions = []

location\_input = input("Enter Location of Vacuum: ")

status\_input = input("Enter status of " + location\_input + ": ")

status\_input\_complement = input("Enter status of other room: ")

print("Initial Location Condition" + str(goal\_state))

if location\_input == 'A':

location\_complement = 'B'

else:

location\_complement = 'A'

if status\_input == '1':

actions.append("Suck at Location "+location\_input)

goal\_state[location\_input] = '0'

cost += 1

actions.append("Move to Location "+location\_complement)

if status\_input\_complement == '1':

cost += 1

actions.append("Suck at Location "+location\_complement)

goal\_state[location\_complement] = '0'

cost += 1

if status\_input == '0':

actions.append("Move to Location "+location\_complement)

if status\_input\_complement == '1':

actions.append("Suck at Location "+location\_complement)

cost += 1

goal\_state[location\_complement] = '0'

cost += 1

print("GOAL STATE: ")

print(goal\_state)

print("Actions Taken are: ")

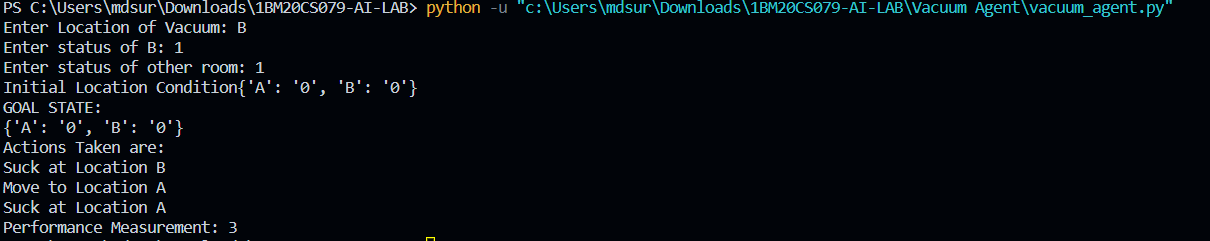
for var in actions:

print(var)

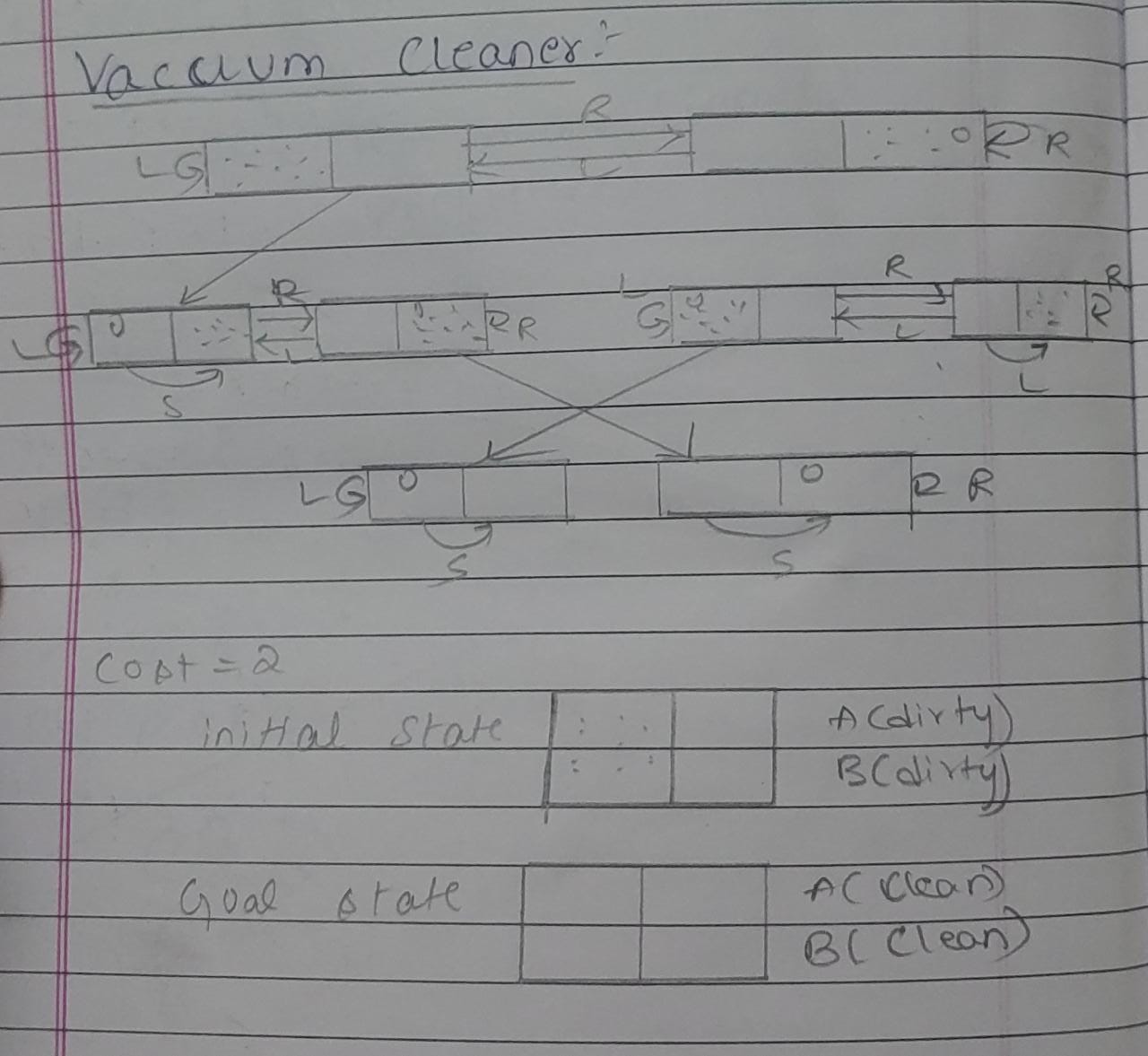
print("Performance Measurement: " + str(cost))

vacuum\_world()

**Output Snapshot**

****

**State Space Tree**

****

**LAB PROGRAM 6**

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

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

**Code:**

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

variable={'p':0,'q':1, 'r':2}

kb=''

q=''

priority={'~':3,'v':1,'^':2}

def input\_rules():

global kb, q

kb = (input("Enter rule: "))

q = input("Enter the Query: ")

def entailment():

global kb, q

print('\*'\*10+"Truth Table Reference"+'\*'\*10)

print('kb','alpha')

print('\*'\*10)

for comb in combinations:

s = evaluatePostfix(toPostfix(kb), comb)

f = evaluatePostfix(toPostfix(q), comb)

print(s, f)

print('-'\*10)

if s and not f:

return False

return True

def isOperand(c):

return c.isalpha() and c!='v'

def isLeftParanthesis(c):

return c == '('

def isRightParanthesis(c):

return c == ')'

def isEmpty(stack):

return len(stack) == 0

def peek(stack):

return stack[-1]

def hasLessOrEqualPriority(c1, c2):

try:

return priority[c1]<=priority[c2]

except KeyError:

return False

def toPostfix(infix):

stack = []

postfix = ''

for c in infix:

if isOperand(c):

postfix += c

else:

if isLeftParanthesis(c):

stack.append(c)

elif isRightParanthesis(c):

operator = stack.pop()

while not isLeftParanthesis(operator):

postfix += operator

operator = stack.pop()

else:

while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):

postfix += stack.pop()

stack.append(c)

while (not isEmpty(stack)):

postfix += stack.pop()

return postfix

def evaluatePostfix(exp, comb):

stack = []

for i in exp:

if isOperand(i):

stack.append(comb[variable[i]])

elif i == '~':

val1 = stack.pop()

stack.append(not val1)

else:

val1 = stack.pop()

val2 = stack.pop()

stack.append(\_eval(i,val2,val1))

return stack.pop()

def \_eval(i, val1, val2):

if i == '^':

return val2 and val1

return val2 or val1

input\_rules()

ans = entailment()

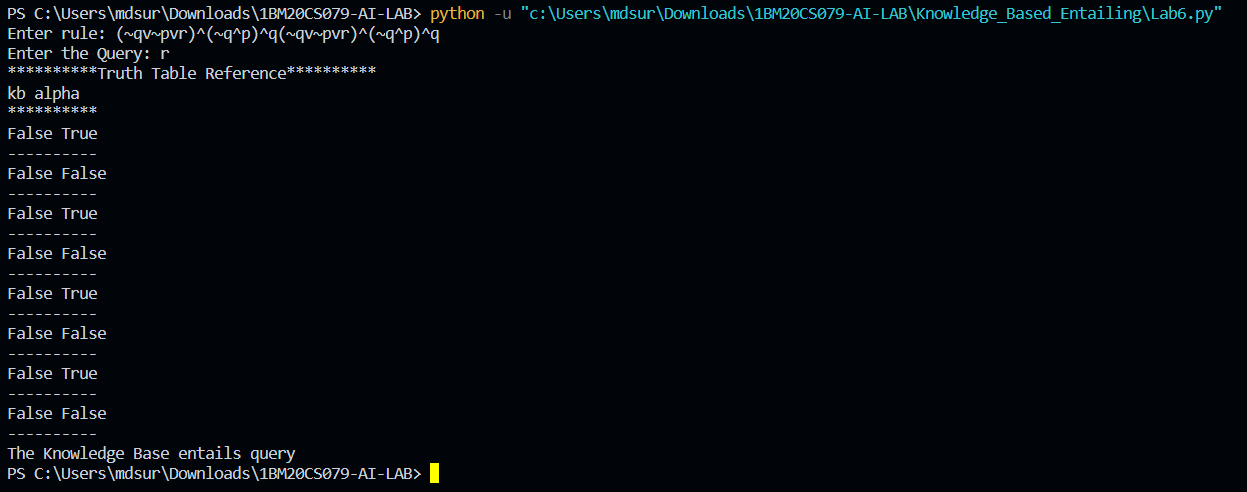
if ans:

print("The Knowledge Base entails query")

else:

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

**OUTPUT SNAPSHOT**

****

**LAB PROGRAM 7**

**Create a knowledge base using prepositional logic and prove the given query using resolution**

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

**Code**

kb = []

# Reset kb to an empty list

def CLEAR():

global kb

kb = []

# Insert sentence to the kb

def TELL(sentence):

global kb

# If the sentence is a clause, insert directly.

if isClause(sentence):

kb.append(sentence)

# If not, convert to CNF, and then insert clauses one by one.

else:

sentenceCNF = convertCNF(sentence)

if not sentenceCNF:

print("Illegal input")

return

# Insert clauses one by one when there are multiple clauses

if isAndList(sentenceCNF):

for s in sentenceCNF[1:]:

kb.append(s)

else:

kb.append(sentenceCNF)

# 'ASK' the kb whether a sentence is True or not

def ASK(sentence):

global kb

# Negate the sentence, and convert it to CNF accordingly.

if isClause(sentence):

neg = negation(sentence)

else:

sentenceCNF = convertCNF(sentence)

if not sentenceCNF:

print("Illegal input")

return

neg = convertCNF(negation(sentenceCNF))

# Insert individual clauses that we need to ask to ask\_list.

ask\_list = []

if isAndList(neg):

for n in neg[1:]:

nCNF = makeCNF(n)

if type(nCNF).\_\_name\_\_ == 'list':

ask\_list.insert(0, nCNF)

else:

ask\_list.insert(0, nCNF)

else:

ask\_list = [neg]

# Create a new list combining the asked sentence and kb.

# Resolution will happen between the items in the list.

clauses = ask\_list + kb[:]

# Recursivly conduct resoltion between items in the clauses list

# until it produces an empty list or there's no more pregress.

while True:

new\_clauses = []

for c1 in clauses:

for c2 in clauses:

if c1 is not c2:

resolved = resolve(c1, c2)

if resolved == False:

continue

if resolved == []:

return True

new\_clauses.append(resolved)

if len(new\_clauses) == 0:

return False

new\_in\_clauses = True

for n in new\_clauses:

if n not in clauses:

new\_in\_clauses = False

clauses.append(n)

if new\_in\_clauses:

return False

return False

# Conduct resolution on two CNF clauses.

def resolve(arg\_one, arg\_two):

resolved = False

s1 = make\_sentence(arg\_one)

s2 = make\_sentence(arg\_two)

resolve\_s1 = None

resolve\_s2 = None

# Two for loops that iterate through the two clauses.

for i in s1:

if isNotList(i):

a1 = i[1]

a1\_not = True

else:

a1 = i

a1\_not = False

for j in s2:

if isNotList(j):

a2 = j[1]

a2\_not = True

else:

a2 = j

a2\_not = False

# cancel out two literals such as 'a' $ ['not', 'a']

if a1 == a2:

if a1\_not != a2\_not:

# Return False if resolution already happend

# but contradiction still exists.

if resolved:

return False

else:

resolved = True

resolve\_s1 = i

resolve\_s2 = j

break

# Return False if not resolution happened

if not resolved:

return False

# Remove the literals that are canceled

s1.remove(resolve\_s1)

s2.remove(resolve\_s2)

# # Remove duplicates

result = clear\_duplicate(s1 + s2)

# Format the result.

if len(result) == 1:

return result[0]

elif len(result) > 1:

result.insert(0, 'or')

return result

# Prepare sentences for resolution.

def make\_sentence(arg):

if isLiteral(arg) or isNotList(arg):

return [arg]

if isOrList(arg):

return clear\_duplicate(arg[1:])

return

# Clear out duplicates in a sentence.

def clear\_duplicate(arg):

result = []

for i in range(0, len(arg)):

if arg[i] not in arg[i+1:]:

result.append(arg[i])

return result

# Check whether a sentence is a legal CNF clause.

def isClause(sentence):

if isLiteral(sentence):

return True

if isNotList(sentence):

if isLiteral(sentence[1]):

return True

else:

return False

if isOrList(sentence):

for i in range(1, len(sentence)):

if len(sentence[i]) > 2:

return False

elif not isClause(sentence[i]):

return False

return True

return False

# Check if a sentence is a legal CNF.

def isCNF(sentence):

if isClause(sentence):

return True

elif isAndList(sentence):

for s in sentence[1:]:

if not isClause(s):

return False

return True

return False

# Negate a sentence.

def negation(sentence):

if isLiteral(sentence):

return ['not', sentence]

if isNotList(sentence):

return sentence[1]

# DeMorgan:

if isAndList(sentence):

result = ['or']

for i in sentence[1:]:

if isNotList(sentence):

result.append(i[1])

else:

result.append(['not', sentence])

return result

if isOrList(sentence):

result = ['and']

for i in sentence[:]:

if isNotList(sentence):

result.append(i[1])

else:

result.append(['not', i])

return result

return None

# Convert a sentence into CNF.

def convertCNF(sentence):

while not isCNF(sentence):

if sentence is None:

return None

sentence = makeCNF(sentence)

return sentence

# Help make a sentence into CNF.

def makeCNF(sentence):

if isLiteral(sentence):

return sentence

if (type(sentence).\_\_name\_\_ == 'list'):

operand = sentence[0]

if isNotList(sentence):

if isLiteral(sentence[1]):

return sentence

cnf = makeCNF(sentence[1])

if cnf[0] == 'not':

return makeCNF(cnf[1])

if cnf[0] == 'or':

result = ['and']

for i in range(1, len(cnf)):

result.append(makeCNF(['not', cnf[i]]))

return result

if cnf[0] == 'and':

result = ['or']

for i in range(1, len(cnf)):

result.append(makeCNF(['not', cnf[i]]))

return result

return "False: not"

# Implication Elimination:

if operand == 'implies' and len(sentence) == 3:

return makeCNF(['or', ['not', makeCNF(sentence[1])], makeCNF(sentence[2])])

# Biconditional Elimination:

if operand == 'biconditional' and len(sentence) == 3:

s1 = makeCNF(['implies', sentence[1], sentence[2]])

s2 = makeCNF(['implies', sentence[2], sentence[1]])

return makeCNF(['and', s1, s2])

if isAndList(sentence):

result = ['and']

for i in range(1, len(sentence)):

cnf = makeCNF(sentence[i])

# Distributivity:

if isAndList(cnf):

for i in range(1, len(cnf)):

result.append(makeCNF(cnf[i]))

continue

result.append(makeCNF(cnf))

return result

if isOrList(sentence):

result1 = ['or']

for i in range(1, len(sentence)):

cnf = makeCNF(sentence[i])

# Distributivity:

if isOrList(cnf):

for i in range(1, len(cnf)):

result1.append(makeCNF(cnf[i]))

continue

result1.append(makeCNF(cnf))

# Associativity:

while True:

result2 = ['and']

and\_clause = None

for r in result1:

if isAndList(r):

and\_clause = r

break

# Finish when there's no more 'and' lists

# inside of 'or' lists

if not and\_clause:

return result1

result1.remove(and\_clause)

for i in range(1, len(and\_clause)):

temp = ['or', and\_clause[i]]

for o in result1[1:]:

temp.append(makeCNF(o))

result2.append(makeCNF(temp))

result1 = makeCNF(result2)

return None

return None

# Below are 4 functions that check the type of a variable

def isLiteral(item):

if type(item).\_\_name\_\_ == 'str':

return True

return False

def isNotList(item):

if type(item).\_\_name\_\_ == 'list':

if len(item) == 2:

if item[0] == 'not':

return True

return False

def isAndList(item):

if type(item).\_\_name\_\_ == 'list':

if len(item) > 2:

if item[0] == 'and':

return True

return False

def isOrList(item):

if type(item).\_\_name\_\_ == 'list':

if len(item) > 2:

if item[0] == 'or':

return True

return False

if \_\_name\_\_ == "\_\_main\_\_":

CLEAR()

print("Test 1")

TELL(['implies', 'p', 'q'])

TELL(['implies', 'r', 's'])

ASK(['implies',['or','p','r'], ['or', 'q', 's']])

CLEAR()

print("Test 2")

TELL('p')

TELL(['implies',['and','p','q'],'r'])

TELL(['implies',['or','s','t'],'q'])

TELL('t')

ASK('r')

CLEAR()

print("Test 3")

TELL('a')

TELL('b')

TELL('c')

TELL('d')

ASK(['or', 'a', 'b', 'c', 'd'])

CLEAR()

print("Test 4")

TELL('a')

TELL('b')

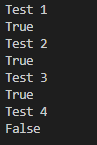
TELL(['or', ['not', 'a'], 'b'])

TELL(['or', 'c', 'd'])

TELL('d')

ASK('c')

**Output Snapshot**

****

**LAB PROGRAM 8**

**Implement unification in first order logic**

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

**Code:**

import re

def getAttributes(expression):

expression = expression.split("(")[1:]

expression = "(".join(expression)

expression = expression.split(")")[:-1]

expression = ")".join(expression)

attributes = expression.split(',')

return attributes

def getInitialPredicate(expression):

return expression.split("(")[0]

def isConstant(char):

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

def isVariable(char):

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

def replaceAttributes(exp, old, new):

attributes = getAttributes(exp)

predicate = getInitialPredicate(exp)

for index, val in enumerate(attributes):

if val == old:

attributes[index] = new

return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):

for substitution in substitutions:

new, old = substitution

exp = replaceAttributes(exp, old, new)

return exp

def checkOccurs(var, exp):

if exp.find(var) == -1:

return False

return True

def getFirstPart(expression):

attributes = getAttributes(expression)

return attributes[0]

def getRemainingPart(expression):

predicate = getInitialPredicate(expression)

attributes = getAttributes(expression)

newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"

return newExpression

def unify(exp1, exp2):

if exp1 == exp2:

return []

if isConstant(exp1) and isConstant(exp2):

if exp1 != exp2:

print(f"{exp1} and {exp2} are constants. Cannot be unified")

return []

if isConstant(exp1):

return [(exp1, exp2)]

if isConstant(exp2):

return [(exp2, exp1)]

if isVariable(exp1):

return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

if isVariable(exp2):

return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

if getInitialPredicate(exp1) != getInitialPredicate(exp2):

print("Cannot be unified as the predicates do not match!")

return []

attributeCount1 = len(getAttributes(exp1))

attributeCount2 = len(getAttributes(exp2))

if attributeCount1 != attributeCount2:

print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be unified")

return []

head1 = getFirstPart(exp1)

head2 = getFirstPart(exp2)

initialSubstitution = unify(head1, head2)

if not initialSubstitution:

return []

if attributeCount1 == 1:

return initialSubstitution

tail1 = getRemainingPart(exp1)

tail2 = getRemainingPart(exp2)

if initialSubstitution != []:

tail1 = apply(tail1, initialSubstitution)

tail2 = apply(tail2, initialSubstitution)

remainingSubstitution = unify(tail1, tail2)

if not remainingSubstitution:

return []

return initialSubstitution + remainingSubstitution

def main():

print("Enter the first expression")

e1 = input()

print("Enter the second expression")

e2 = input()

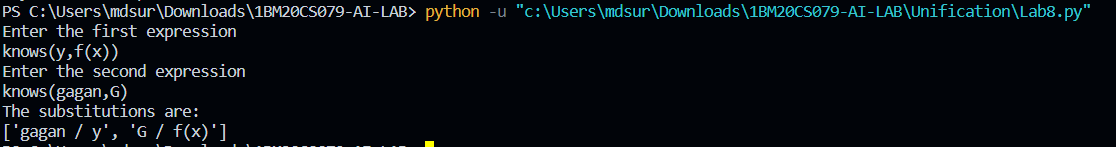
substitutions = unify(e1, e2)

print("The substitutions are:")

print([' / '.join(substitution) for substitution in substitutions])

main()

**Output Snapshot**

****

**LAB PROGRAM 9**

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

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

**Code:**

import re

def getAttributes(string):

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

matches = re.findall(expr, string)

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

def getPredicates(string):

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

return re.findall(expr, string)

def DeMorgan(sentence):

string = ''.join(list(sentence).copy())

string = string.replace('~~','')

flag = '[' in string

string = string.replace('~[','')

string = string.strip(']')

for predicate in getPredicates(string):

string = string.replace(predicate, f'~{predicate}')

s = list(string)

for i, c in enumerate(string):

if c == 'V':

s[i] = '^'

elif c == '^':

s[i] = 'V'

string = ''.join(s)

string = string.replace('~~','')

return f'[{string}]' if flag else string

def Skolemization(sentence):

SKOLEM\_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]

statement = ''.join(list(sentence).copy())

matches = re.findall('[∀∃].', statement)

for match in matches[::-1]:

statement = statement.replace(match, '')

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

for s in statements:

statement = statement.replace(s, s[1:-1])

for predicate in getPredicates(statement):

attributes = getAttributes(predicate)

if ''.join(attributes).islower():

statement = statement.replace(match[1],SKOLEM\_CONSTANTS.pop(0))

else:

aL = [a for a in attributes if a.islower()]

aU = [a for a in attributes if not a.islower()][0]

statement = statement.replace(aU, f'{SKOLEM\_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')

return statement

def fol\_to\_cnf(fol):

statement = fol.replace("<=>", "\_")

while '\_' in statement:

i = statement.index('\_')

new\_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^['+ statement[i+1:] + '=>' + statement[:i] + ']'

statement = new\_statement

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

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

statements = re.findall(expr, statement)

for i, s in enumerate(statements):

if '[' in s and ']' not in s:

statements[i] += ']'

for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s))

while '-' in statement:

i = statement.index('-')

br = statement.index('[') if '[' in statement else 0

new\_statement = '~' + statement[br:i] + 'V' + statement[i+1:]

statement = statement[:br] + new\_statement if br > 0 else new\_statement

while '~∀' in statement:

i = statement.index('~∀')

statement = list(statement)

statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'

statement = ''.join(statement)

while '~∃' in statement:

i = statement.index('~∃')

s = list(statement)

s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'

statement = ''.join(s)

statement = statement.replace('~[∀','[~∀')

statement = statement.replace('~[∃','[~∃')

expr = '(~[∀V∃].)'

statements = re.findall(expr, statement)

for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s))

expr = '~\[[^]]+\]'

statements = re.findall(expr, statement)

for s in statements:

statement = statement.replace(s, DeMorgan(s))

return statement

def main():

print("Enter FOL:")

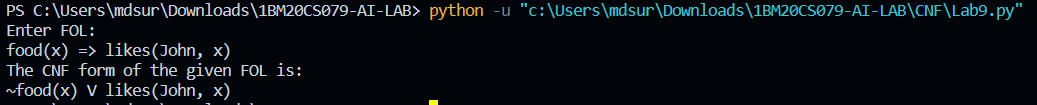
fol = input()

print("The CNF form of the given FOL is: ")

print(Skolemization(fol\_to\_cnf(fol)))

main()

**Output Snapshot**

****

**LAB PROGRAM 10**

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

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

**Code**

import re

def isVariable(x):

return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):

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

matches = re.findall(expr, string)

return matches

def getPredicates(string):

expr = '([a-z~]+)\([^&|]+\)'

return re.findall(expr, string)

class Fact:

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

self.expression = expression

predicate, params = self.splitExpression(expression)

self.predicate = predicate

self.params = params

self.result = any(self.getConstants())

def splitExpression(self, expression):

predicate = getPredicates(expression)[0]

params = getAttributes(expression)[0].strip('()').split(',')

return [predicate, params]

def getResult(self):

return self.result

def getConstants(self):

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

def getVariables(self):

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

def substitute(self, constants):

c = constants.copy()

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

return Fact(f)

class Implication:

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

self.expression = expression

l = expression.split('=>')

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

self.rhs = Fact(l[1])

def evaluate(self, facts):

constants = {}

new\_lhs = []

for fact in facts:

for val in self.lhs:

if val.predicate == fact.predicate:

for i, v in enumerate(val.getVariables()):

if v:

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

new\_lhs.append(fact)

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

for key in constants:

if constants[key]:

attributes = attributes.replace(key, constants[key])

expr = f'{predicate}{attributes}'

return Fact(expr) if len(new\_lhs) and all([f.getResult() for f in new\_lhs]) else None

class KB:

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

self.facts = set()

self.implications = set()

def tell(self, e):

if '=>' in e:

self.implications.add(Implication(e))

else:

self.facts.add(Fact(e))

for i in self.implications:

res = i.evaluate(self.facts)

if res:

self.facts.add(res)

def 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}')

def main():

kb = KB()

print("Enter KB: (enter e to exit)")

while True:

t = input()

if(t == 'e'):

break

kb.tell(t)

print("Enter Query:")

q = input()

kb.query(q)

kb.display()

main()

**Output Snapshot**

****