**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**ARTIFICIAL INTELLIGENCE**

***Submitted by***

**ADVITHI D  (1BM21CS009)**

***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**

**October-2023 to Feb-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 Lab work entitled “Artificial Intelligence” carried out by **Advithi D (1BM21CS009),** who is bonafide student of **B. M. S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the year 2023.  The Lab report has been approved as it satisfies the academic requirements in respect of a **Artificial Intelligence** course **(22CS5PCAIN)** work prescribed for the said degree.

**Gauri Kalnoor                            Dr. Jyothi S Nayak**

Assistant Professor                                  Professor and Head

Department of CSE                   Department of CSE

BMSCE, Bengaluru                   BMSCE, Bengaluru

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl. No.** | **Date** | **Experiment Title** | **Page No.** |
| **01** | 25-11-23 | Implement Tic –Tac –Toe Game. | 1-3 |
| **02** | 12-12-23 | Solve 8 puzzle problems. | 4-6 |
| **03** | 12-12-23 | Implement Iterative deepening search algorithm. | 7-8 |
| **04** | 19-12-23 | Implement A\* search algorithm. | 9-11 |
| **05** | 26-12-23 | Write a program to implement Simulated Annealing Algorithm. | 12 |
| **06** | 09-01-24 | Implement vacuum cleaner agent. | 13-14 |
| **07** | 16-01-24 | Create a knowledge base using prepositional logic and show that the given query entails the knowledge base or not. | 15-16 |
| **08** | 23-01-24 | Create a knowledge base using prepositional logic and prove the given query using resolution. | 17-20 |
| **09** | 23-01-24 | Implement unification in first order logic. | 21-24 |
| **10** | 23-01-24 | Convert a given first order logic statement into Conjunctive Normal Form (CNF). | 25-26 |
| **11** | 23-01-24 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 27-30 |

**1) Implement Tic –Tac –Toe Game.**

import numpy as np

import random

from time import sleep

def create\_board():

return(np.array([[0, 0, 0], [0, 0, 0], [0, 0, 0]]))

def possibilities(board):

l = []

for i in range(len(board)):

for j in range(len(board)):

if board[i][j] == 0:

l.append((i, j))

return(l)

def random\_place(board, player):

selection = possibilities(board)

current\_loc = random.choice(selection)

board[current\_loc] = player

return(board)

def row\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[x, y] != player:

win = False

continue

if win == True:

return(win)

return(win)

def col\_win(board, player):

for x in range(len(board)):

win = True

for y in range(len(board)):

if board[y][x] != player:

win = False

continue

if win == True:

return(win)

return(win)

def diag\_win(board, player):

win = True

y = 0

for x in range(len(board)):

if board[x, x] != player:

win = False

if win:

return win

win = True

if win:

for x in range(len(board)):

y = len(board) - 1 - x

if board[x, y] != player:

win = False

return win

def evaluate(board):

winner = 0

for player in [1, 2]:

if (row\_win(board, player) or

col\_win(board, player) or

diag\_win(board, player)):

winner = player

if np.all(board != 0) and winner == 0:

winner = -1

return winner

def play\_game():

board, winner, counter = create\_board(), 0, 1

print(board)

sleep(2)

while winner == 0:

for player in [1, 2]:

board = random\_place(board, player)

print("Board after " + str(counter) + " move")

print(board)

sleep(2)

counter += 1

winner = evaluate(board)

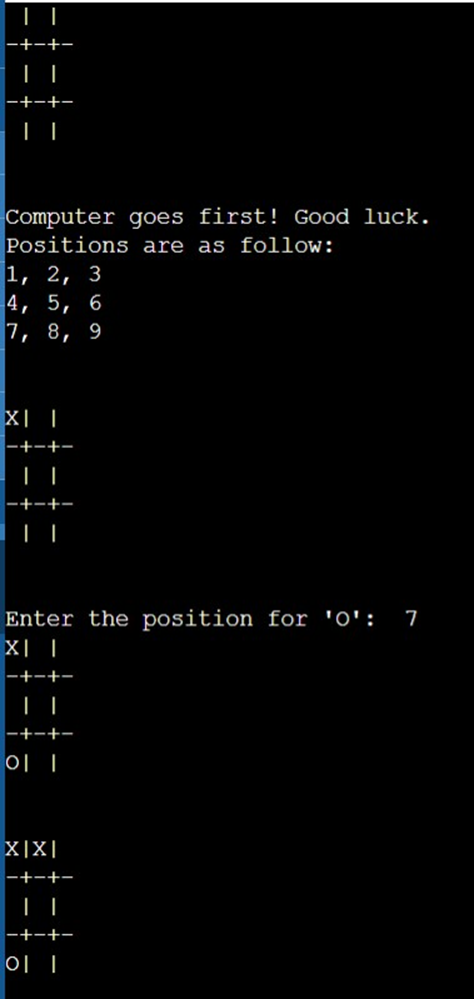
if winner != 0:

break

return(winner)

print("Winner is: " + str(play\_game()))

OUTPUT



**2) Solve 8 puzzle problems.**

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

    def \_\_init\_\_(self):

        self.heap = []

    def push(self, key):

        heappush(self.heap, key)

    def pop(self):

        return heappop(self.heap)

    def empty(self):

        if not self.heap:

            return True

        else:

            return False

class nodes:

    def \_\_init\_\_(self, parent, mats, empty\_tile\_posi,

                costs, levels):

        self.parent = parent

        self.mats = mats

          self.empty\_tile\_posi = empty\_tile\_posi

        self.costs = costs

        self.levels = levels

    def \_\_lt\_\_(self, nxt):

        return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

    count = 0

    for i in range(n):

        for j in range(n):

            if ((mats[i][j]) and

                (mats[i][j] != final[i][j])):

                count += 1

    return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

            levels, parent, final) -> nodes:

    new\_mats = copy.deepcopy(mats)

    x1 = empty\_tile\_posi[0]

    y1 = empty\_tile\_posi[1]

    x2 = new\_empty\_tile\_posi[0]

    y2 = new\_empty\_tile\_posi[1]

    new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

    costs = calculateCosts(new\_mats, final)

      new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

                    costs, levels)

    return new\_nodes

def printMatsrix(mats):

    for i in range(n):

        for j in range(n):

            print("%d " % (mats[i][j]), end = " ")

        print()

def isSafe(x, y):

    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):

    if root == None:

      return

    printPath(root.parent)

    printMatsrix(root.mats)

    print()

def solve(initial, empty\_tile\_posi, final):

    pq = priorityQueue()

    costs = calculateCosts(initial, final)

    root = nodes(None, initial,

                empty\_tile\_posi, costs, 0)

    pq.push(root)

    while not pq.empty():

      minimum = pq.pop()

        if minimum.costs == 0:

            printPath(minimum)

            return

        for i in range(n):

            new\_tile\_posi = [

                minimum.empty\_tile\_posi[0] + rows[i],

                minimum.empty\_tile\_posi[1] + cols[i], ]

                              if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

                child = newNodes(minimum.mats,

                                minimum.empty\_tile\_posi,

                                new\_tile\_posi,

                                minimum.levels + 1,

       minimum, final,)

     pq.push(child)

initial = [ [ 1, 2, 3 ],

            [ 5, 6, 0 ],

            [ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

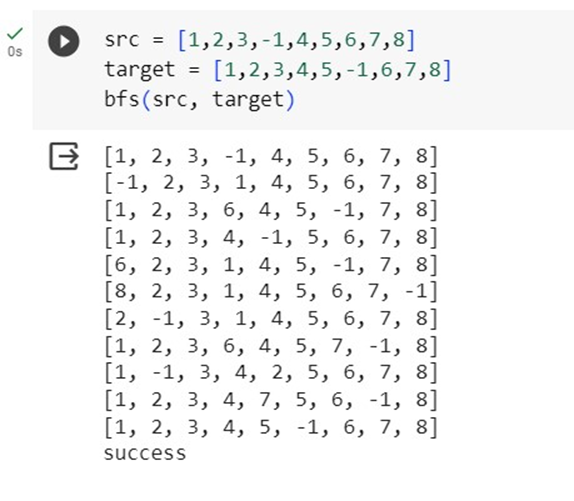
        [ 5, 8, 6 ],

        [ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

OUTPUT



**3) Implement Iterative deepening search algorithm.**

from collections import defaultdict

class Graph:

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

self.V = vertices

self.graph = defaultdict(list)

def addEdge(self,u,v):

self.graph[u].append(v)

def DLS(self,src,target,maxDepth):

if src == target : return True

if maxDepth <= 0 : return False

for i in self.graph[src]:

if(self.DLS(i,target,maxDepth-1)):

return True

return False

def IDDFS(self,src, target, maxDepth):

for i in range(maxDepth):

if (self.DLS(src, target, i)):

return True

return False

g = Graph (7);

g.addEdge(0, 1)

g.addEdge(0, 2)

g.addEdge(1, 3)

g.addEdge(1, 4)

g.addEdge(2, 5)

g.addEdge(2, 6)

target = 6; maxDepth = 3; src = 0

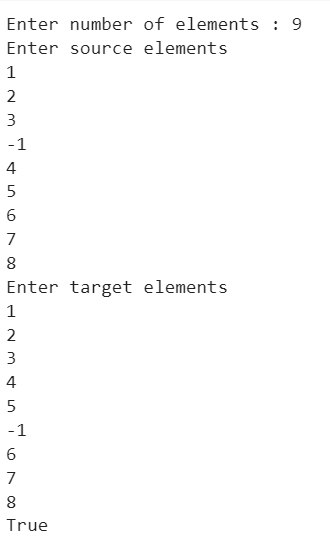
if g.IDDFS(src, target, maxDepth) == True:

print ("Target is reachable from source within max depth")

else :

print ("Target is NOT reachable from source within max depth")

OUTPUT



**4) Implement A\* search algorithm.**

from collections import deque

class Graph:

def \_\_init\_\_(self, adjac\_lis):

self.adjac\_lis = adjac\_lis

def get\_neighbors(self, v):

return self.adjac\_lis[v]

def h(self, n):

H = {

'A': 1,

'B': 1,

'C': 1,

'D': 1

}

return H[n]

def a\_star\_algorithm(self, start, stop):

open\_lst = set([start])

closed\_lst = set([])

poo = {}

poo[start] = 0

par = {}

par[start] = start

while len(open\_lst) > 0:

n = None

for v in open\_lst:

if n == None or poo[v] + self.h(v) < poo[n] + self.h(n):

n = v;

if n == None:

print('Path does not exist!')

return None

if n == stop:

reconst\_path = []

while par[n] != n:

reconst\_path.append(n)

n = par[n]

reconst\_path.append(start)

reconst\_path.reverse()

print('Path found: {}'.format(reconst\_path))

return reconst\_path

for (m, weight) in self.get\_neighbors(n):

if m not in open\_lst and m not in closed\_lst:

open\_lst.add(m)

par[m] = n

poo[m] = poo[n] + weight

else:

if poo[m] > poo[n] + weight:

poo[m] = poo[n] + weight

par[m] = n

if m in closed\_lst:

closed\_lst.remove(m)

open\_lst.add(m)

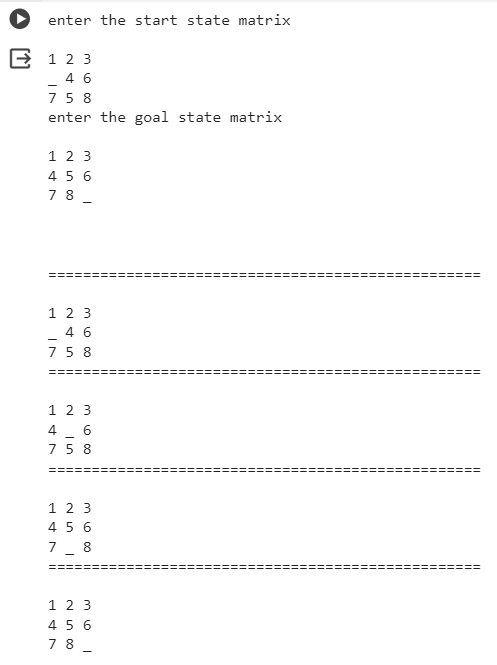
open\_lst.remove(n)

closed\_lst.add(n)

print('Path does not exist!')

return None

OUTPUT



**5) Write a program to implement Simulated Annealing Algorithm**

import math

import random

def objective\_function(x):

return math.sin(x) \* (1 + 0.1 \* x)

def simulated\_annealing(initial\_solution, temperature, cooling\_rate, max\_iterations):

current\_solution = initial\_solution

current\_energy = objective\_function(current\_solution)

for iteration in range(max\_iterations):

temperature \*= cooling\_rate

neighbor\_solution = current\_solution + random.uniform(-1, 1)

neighbor\_energy = objective\_function(neighbor\_solution)

energy\_difference = neighbor\_energy - current\_energy

if energy\_difference < 0 or random.uniform(0, 1) < math.exp(-energy\_difference / temperature):

current\_solution = neighbor\_solution

current\_energy = neighbor\_energy

return current\_solution, current\_energy

initial\_solution = 2.0

initial\_temperature = 1.0

cooling\_rate = 0.95

max\_iterations = 1000

final\_solution, final\_energy = simulated\_annealing(initial\_solution, initial\_temperature, cooling\_rate, max\_iterations)

print(f"Final Solution: {final\_solution}")

print(f"Final Energy: {final\_energy}")

OUTPUT:



**6)Implement vaccum cleaner agent.**

import random

def display(room):

print(room)

room = [

[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1],

[1, 1, 1, 1],

]

print("All the rooom are dirty")

display(room)

x =0

y= 0

while x < 4:

while y < 4:

room[x][y] = random.choice([0,1])

y+=1

x+=1

y=0

print("Before cleaning the room I detect all of these random dirts")

display(room)

x =0

y= 0

z=0

while x < 4:

while y < 4:

if room[x][y] == 1:

print("Vaccum in this location now,",x, y)

room[x][y] = 0

print("cleaned", x, y)

z+=1

y+=1

x+=1

y=0

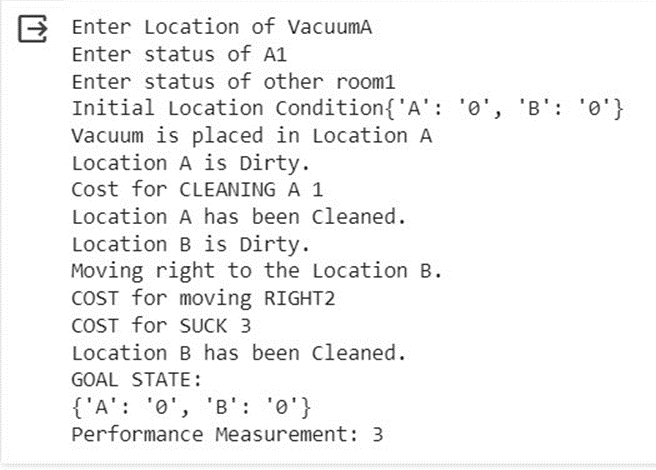
pro= (100-((z/16)\*100))

print("Room is clean now, Thanks for using : 3710933")

display(room)

print('performance=',pro,'%')

OUTPUT



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

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]): ")

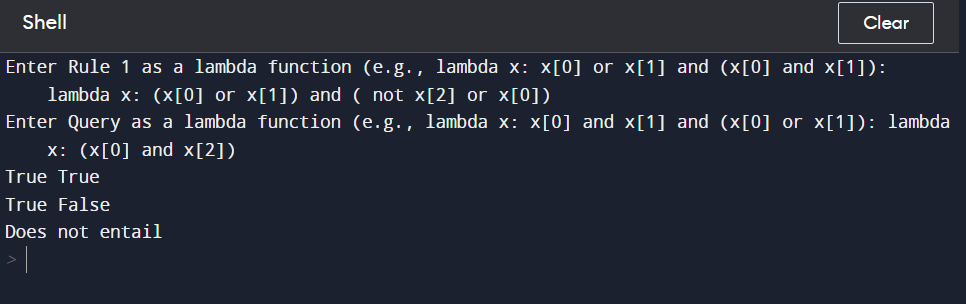
q = eval(query\_str)

# Ask KB Query

result = ask(kb, q)

print(result)

**OUTPUT:**

****

**8)** **Create a knowledge base 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

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

goal = 'R'

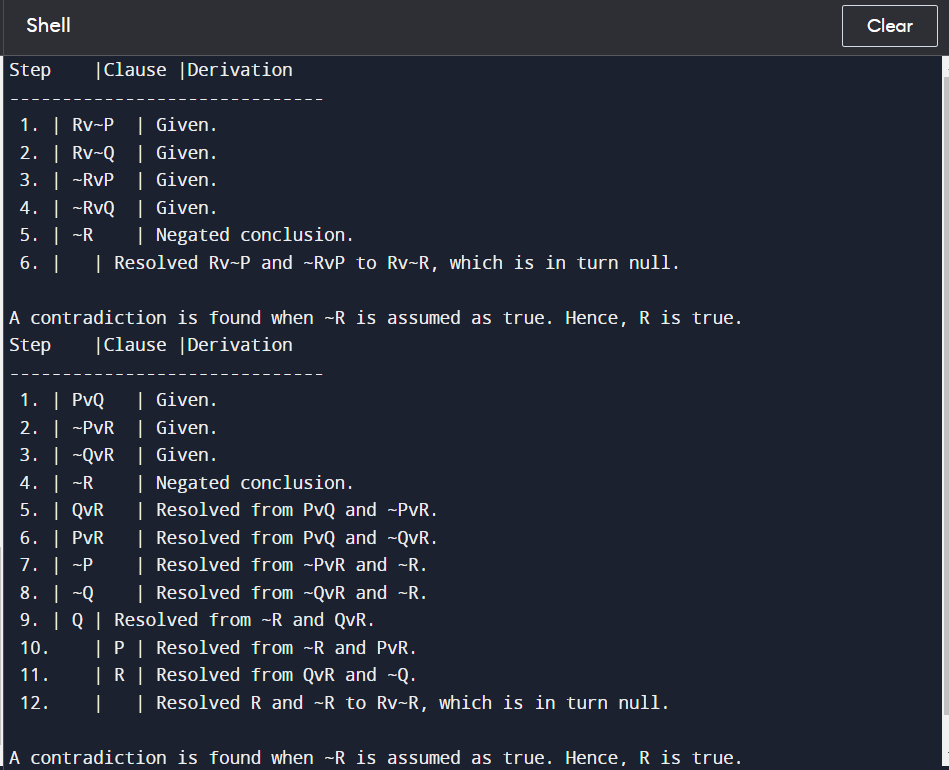
main(rules, goal)

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

goal = 'R'

main(rules, goal)

**OUTPUT:**

****

**9)** **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

exp1 = "knows(A,x)"

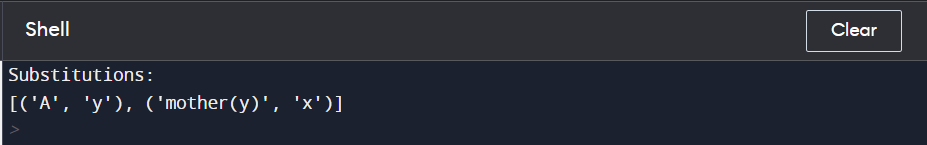
exp2 = "knows(y,mother(y))"

substitutions = unify(exp1, exp2)

print("Substitutions:")

print(substitutions)

**OUTPUT:**



**10)** **Convert a given first order logic statement into Conjunctive Normal Form (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:]

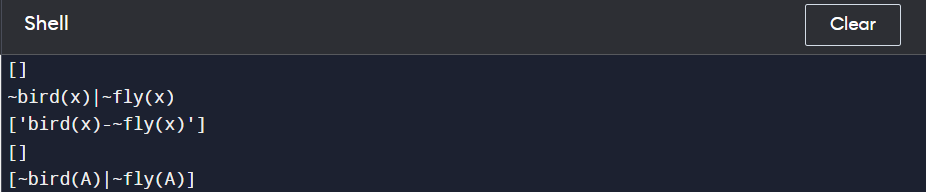
statement = statement[:br] + new\_statement if br > 0 else new\_statement

return Skolemization(statement)

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

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

**OUTPUT:**

****

**11)** **Create a knowledge base 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}')

kb\_ = KB()

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

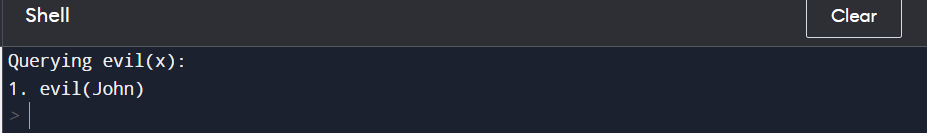
kb\_.tell('king(John)')

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

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

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

**OUTPUT:**

****