KB_ENTAILMENT

January 5, 2022

[35]:
```
combination=[(True,True,True),(True,True,False),(True,False,True),(True,False,False),(False,Tr
```

[36]:
```
variable={'p':0,'q':1,'r':2}
```

[37]:
```
kb=''
```

[38]:
```
q=''
```

[39]:
```
priority={'~':3,'v':1,'^':2}
```

[40]:
```
def input_rules():
    global kb,q
    kb=input('Enter the rule: ')
    q=input('Enter the query: ')
```

[52]:
```
def entailment():
    global kb,q
    print("kb","alpha")
    print('*' * 10)
    print('POSTFIX OF KB IS ' + str(topostfix(kb)))
    print('POSTFIX OF THE QUERY IS ' + str(topostfix(q)))
    for comb in combination:



        s=evaluate_postfix(topostfix(kb),comb)

        f=evaluate_postfix(topostfix(q),comb)
        print(s,f)



        if s and not f:
            return False
    return True
```

[53]:
```
def isoperand(c):
    return c.isalpha() and c!='v'
```

[54]:
```
def isleftparenthesis(c):
    return c=='('
```

[55]:
```
def isrightparenthesis(c):
    return c==')'
```

[56]:
```
def isEmpty(stack):
    return len(stack)==0
```

[57]:
```
def peek(stack):
    return stack[-1]
```

[58]:
```
def has_less_or_equal_priority(c1,c2):
    try:
        return priority[c1]<=priority[c2]
    except KeyError: return False
```

[59]:
```
def topostfix(infix):
    stack=[]
    postfix=''
    for c in infix:
        if isoperand(c):
            postfix+=c
        else:
            if isleftparenthesis(c):
                stack.append(c)
            elif isrightparenthesis(c):
                operator=stack.pop()
                while not isleftparenthesis(operator):
                    postfix+=operator
                    operator=stack.pop()
            else:
                while (not isEmpty(stack) and␣
↪has_less_or_equal_priority(c,peek(stack))):
                    postfix+=stack.pop()
                stack.append(c)
    while(not isEmpty(stack)):
        postfix+=stack.pop();
    return postfix
```

[60]:
```
def _eval(i,val1,val2):
    if i=='^':
        return val1 and val2
    return val1 or val2
```

[61]:
```
def evaluate_postfix(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,val1,val2))
    return stack.pop();
```

[62]:
```
input_rules()
ans = entailment()
if ans: print("The Knowledge Base Entails Query")
else: print("The Knowledge Base Doesn't Entail Query")
```

```
Enter the rule: (~qv~pvr)^(~q^p)^q
Enter the query: r
kb alpha
**********
POSTFIX OF KB IS q~p~vrvq~p^^q^
POSTFIX OF THE QUERY IS r
False True
False False
False True
False False
False True
False False
False True
False False
The Knowledge Base Entails Query
```

[ ]:

KB_RESOLUTION

January 6, 2022

[31]:
```
def disjunctify(clauses):
    disjuncts=[]
    for clause in clauses:
        disjuncts.append(tuple(clause.split('v')))
    return disjuncts
```

[32]:
```
def getresolvant(ci,cj,di,dj):
    resolvant=list(ci) + list(cj)
    resolvant.remove(di)
    resolvant.remove(dj)
    return tuple(resolvant)
```

[33]:
```
def resolve(ci,cj):
    for di in ci:
        for dj in cj:
            if di == '~' + dj or dj == '~' + di:
                return getresolvant(ci,cj,di,dj)
```

[38]:
```
def check_resolution(clauses,query):
    clauses+=[query if query.startswith('~') else '~' + query]
    proposition='^'.join(['(' + clause + ')' for clause in clauses])
    print(f'Trying to prove {proposition} by contradiction...')
    clauses=disjunctify(clauses)
    resolved=False
    new=set()


    while not resolved:
        n=len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i +␣
↪1, n)]
        for (ci,cj) in pairs:
            resolvant=resolve(ci,cj)
            if not resolvant:
                resolved=True
                break
```

```
            new=new.union(set(resolvant))
        if new.issubset(set(clauses)):
            break
        for clause in new:
            if clause not in clauses:
                clauses.append(clause)

    if resolved:
        print('SUCCESSFULLY PROVED BY RESOLUTION')
    else:
        print('CANNOT BE PROVED')
```

[39]:
```
clauses = input('Enter the clauses ').split()
query = input('Enter the query: ')
check_resolution(clauses, query)
```

```
Enter the clauses AvB BvC ~C
Enter the query: B
Trying to prove (AvB)^(BvC)^(~C)^(~B) by contradiction…
SUCCESSFULLY PROVED BY RESOLUTION
```

[43]:
```
clauses = input('Enter the clauses ').split()
query = input('Enter the query: ')
check_resolution(clauses, query)
```

```
Enter the clauses ~Qv~PvR ~Q^P Q
Enter the query: Q
Trying to prove (~Qv~PvR)^(~Q^P)^(Q)^(~Q) by contradiction…
SUCCESSFULLY PROVED BY RESOLUTION
```

[ ]:
```

```

FINAL_UNIFICATION

January 12, 2022

[495]:
```
predicate=[]
```

[496]:
```
no_of_arg=[]
```

[497]:
```
arguement=[]
```

[498]:
```
no_of_predicat=0
```

[499]:
```
expressions=[]
```

[500]:
```
fu=[]
```

[501]:
```
lst=[]
```

[502]:
```
def function1():
    print('------PROGRAM FOR UNIFICATION---------')
    no_of_predicat=int(input('Enter the number of predicate '))
    for i in range(no_of_predicat):
        expr=input('Enter the expression ' + str(i) + " ")
        expressions.append(expr)
    for exp in expressions:
        fu=exp.split('(')
        one=fu[0]
        predicate.append(one)
        lst=fu[1]
        lst=lst[:-1]
        lst=lst.split(',')

        if lst:
            arguement.append(lst)
    check_arg(no_of_predicat)
```

[503]:
```
def check_arg(no_of_predicat):

    pred_flag=0
```

```
    arg_flag=0
    print('---------CHCEKING PREDICATES----------')

    for i in range(no_of_predicat-1):

        if predicate[i]!=predicate[i+1]:
            print('PREDICAT NOT SAME')
            print('CANNOT BE UNIFIED')
            pred_flag=1
            break

    if pred_flag!=1:
        for i in range(no_of_predicat-1):
            len1=len(arguement[i])
            len2=len(arguement[i+1])
            if len1!=len2:
                print('Arguement are not same cannot be unified')
                arg_flag=1
                break

    if(arg_flag==0 and pred_flag==0):
        unify(no_of_predicat)
```

[504]:
```
def unify(no_of_predicat):

    flag=0
    for i in range(no_of_predicat-1):

        for j in range(len(arguement[i])):

            if arguement[i][j]!=arguement[i+1][j]:

                print(' SUBSTITUTE ' + str(arguement[i][j]) + '/' +␣
↪str(arguement[i+1][j]))
                arguement[i][j]=arguement[i+1][j]
                flag+=1
    if flag==0:
        print(' AREGUEMENT ARE IDENTICAL NO SUBSTITUTION REQUIRED ')
        print(' NO NEED OF SUBSTITUTION ')
```

[493]:
```
function1()
```

```
------PROGRAM FOR UNIFICATION---------
Enter the number of predicate 2
```

```
Enter the expression 0 king(x,john)
Enter the expression 1 queen(y,jane)
---------CHCEKING PREDICATES-----------
PREDICAT NOT SAME
CANNOT BE UNIFIED
```

[505]:
```
function1()
```

```
------PROGRAM FOR UNIFICATION---------
Enter the number of predicate 2
Enter the expression 0 knows(john,x)
Enter the expression 1 knows(y,bill)
---------CHCEKING PREDICATES-----------
 SUBSTITUTE john/y
 SUBSTITUTE x/bill
```

[482]:
```
function1()
```

```
------PROGRAM FOR UNIFICATION---------
Enter the number of predicate 2
Enter the expression 0 studies(x,g(x))
Enter the expression 1 studies(z,g(z))
---------CHCEKING PREDICATES-----------
2
range(0, 1)
 SUBSTITUTE x/z
```

[ ]:
```

```

FOL_TO_CNF

January 13, 2022

[1]:
```
def getAttributes(string):
    expr = '\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

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

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

def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
    matches = re.findall('[ ].', statement)
    for match in matches[::-1]:
        statement = statement.replace(match, '')
        statements = re.findall('\[\[^]]+\]]', statement)
        for s in statements:
            statement = statement.replace(s, s[1:-1])
        for predicate in getPredicates(statement):
```

```
            attributes = getAttributes(predicate)
            if ''.join(attributes).islower():
                statement = statement.replace(match[1],SKOLEM_CONSTANTS.pop(0))
            else:
                aL = [a for a in attributes if a.islower()]
                aU = [a for a in attributes if not a.islower()][0]
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.
↪pop(0)}({aL[0] if len(aL) else match[1]})')
    return statement
```

[2]:
```
import re

def fol_to_cnf(fol):

    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']&['+␣
↪statement[i+1:] + '=>' + statement[:i] + ']'
        statement = new_statement
    statement = statement.replace("=>", "-")
    expr = '\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + '|' + statement[i+1:]
        statement = statement[:br] + new_statement if br > 0 else new_statement
    while '~ ' in statement:
        i = statement.index('~ ')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = ' ', statement[i+2], '~'
        statement = ''.join(statement)
    while '~ ' in statement:
        i = statement.index('~ ')
        s = list(statement)
        s[i], s[i+1], s[i+2] = ' ', s[i+2], '~'
        statement = ''.join(s)
    statement = statement.replace('~[ ','[~ ')
    statement = statement.replace('~[ ','[~ ')
    expr = '(~[ | ].)'
    statements = re.findall(expr, statement)
```

```
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))
    expr = '~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))
    return statement
```

[3]:
```
n = int(input())
while n:
  statement = input("Enter FOL statement: ")
  print(f"FOL converted to CNF: {Skolemization(fol_to_cnf(statement))} \n\n")
  n -= 1
```

```
3
Enter FOL statement:
[american(x)&weapon(y)&sells(x,y,z)&hostile(z)]=>criminal(x)
FOL converted to CNF:
[~american(x)|~weapon(y)|~sells(x,y,z)|~hostile(z)]|criminal(x)


Enter FOL statement:  x[ y[animal(y)=>loves(x,y)]]=>[ z[loves(z,x)]]
FOL converted to CNF: [animal(G(x))&~loves(x,G(x))]|[loves(F(x),x)]


Enter FOL statement: animal(y)<=>loves(x,y)
FOL converted to CNF: [~animal(y)|loves(x,y)]&[~loves(x,y)|animal(y)]
```

[ ]:

FOL_FORWARD_REASONING

January 13, 2022

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

[1]:
```
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)
```

[2]:
```
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}')
```

[3]:
```
kb = KB()
kb.tell('missile(x)=>weapon(x)')
kb.tell('missile(M1)')
kb.tell('enemy(x,America)=>hostile(x)')
kb.tell('american(West)')
kb.tell('enemy(Nono,America)')
kb.tell('owns(Nono,M1)')
kb.tell('missile(x)&owns(Nono,x)=>sells(West,x,Nono)')
kb.tell('american(x)&weapon(y)&sells(x,y,z)&hostile(z)=>criminal(x)')
kb.query('criminal(x)')
kb.display()
```

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

[ ]: