SAKSHI ROY

1BM19CS140

5-C

AI LAB REPORT

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

**def** processRule(rule):

rule **=** rule**.**replace('~', ' not ')

rule **=** rule**.**replace('^', ' and ')

rule **=** rule**.**replace('v', ' or ')

**return** rule

**def** formatRule(rule, P, Q, R):

P, Q, R **=** str(P), str(Q), str(R)

rule **=** rule**.**replace('P', P)

rule **=** rule**.**replace('Q', Q)

rule **=** rule**.**replace('R', R)

**return** rule

**def** checkEntailment(rule, query):

models **=** [

(**False**, **False**, **False**),

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

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

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

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

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

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

(**True**, **True**, **True**)

]

rule **=** processRule(rule)

entails **=** **True**

**for** P, Q, R **in** models:

formattedRule **=** formatRule(rule, P, Q, R)

print(f'Evaluating: {formattedRule}')

KB **=** eval(formattedRule)

\_query **=** R **if** query **==** 'R' **else** P **if** query **==** 'P' **else** Q

print(f'Knowledge Base: {KB} Query: {\_query}')

**if** KB:

entails **&=** KB **and** \_query

**if** entails:

print('Knowledge Base entails the query')

**else**:

print("Knowledge Base doesn't entail the query")

rule **=** input('Enter the rule: ')

query **=** input('Enter the query: ')

*#R*

checkEntailment(rule, query)

rule **=** input('Enter the rule: ')

*#(PvQ)^(~RvP)*

query **=** input('Enter the query: ')

*#R*

checkEntailment(rule, query)

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

**def** disjunctify(clauses):

disjuncts **=** []

**for** clause **in** clauses:

disjuncts**.**append(tuple(clause**.**split('v')))

**return** disjuncts

**def** getResolvant(ci, cj, di, dj):

resolvant **=** list(ci) **+** list(cj)

resolvant**.**remove(di)

resolvant**.**remove(dj)

**return** tuple(resolvant)

**def** resolve(ci, cj):

**for** di **in** ci:

**for** dj **in** cj:

**if** di **==** '~' **+** dj **or** dj **==** '~' **+** di:

**return** getResolvant(ci, cj, di, dj)

**def** checkResolution(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(resolvents))

**if** new**.**issubset(set(clauses)):

**break**

**for** clause **in** new:

**if** clause **not** **in** clauses:

clauses**.**append(clause)

**if** resolved:

print('Knowledge Base entails the query, proved by resolution')

**else**:

print("Knowledge Base doesn't entail the query, no empty set produced after resolution")

clauses **=** input('Enter the clauses separated by a space: ')**.**split()

*# AvB BvC ~C*

query **=** input('Enter the query: ')

*# A*

checkResolution(clauses, query)

clauses **=** input('Enter the clauses separated by a space: ')**.**split()

*# ~Qv~PvR ~Q^P Q*

query **=** input('Enter the query: ')

*# R*

checkResolution(clauses, query)

3. Implement unification in first order logic.

**def** getAttributes(expression):

'''Return the attributes of a predicate'''

expression **=** expression**.**split("(")[1:] *# Remove the first '(' after the predicate*

expression **=** "("**.**join(expression)

expression **=** expression**.**split(")")[:**-**1] *# Remove the last ')' after the last attribute*

expression **=** ")"**.**join(expression)

attributes **=** expression**.**split(',')

**return** attributes

**def** getPredicate(expression):

'''Return the predicate'''

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

**def** isConstant(char):

'''Upper case characters are assumed to be constants'''

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

**def** isVariable(char):

'''Lower case characters are assumed to be variables'''

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

**def** substitute(exp, old, new):

'''Perform substitution of the old attribute in the expression'''

attributes **=** getAttributes(exp)

predicate **=** getPredicate(exp)

**for** index, val **in** enumerate(attributes):

**if** val **==** old:

attributes[index] **=** new

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

**def** apply(exp, substitutions):

'''Apply the substitutions in the expression'''

**for** substitution **in** substitutions:

new, old **=** substitution

exp **=** substitute(exp, old, new)

**return** exp

**def** checkOccurs(var, exp):

'''Check if variable occurs in the expression'''

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

**return** **False**

**return** **True**

**def** getFirstAttribute(expression):

'''Return the first attribute in the predicate'''

attributes **=** getAttributes(expression)

**return** attributes[0]

**def** getRemaining(expression):

'''Return the expression excluding the first attribute in the predicate'''

predicate **=** getPredicate(expression)

attributes **=** getAttributes(expression)

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

**return** newExpression

**def** unify(exp1, exp2):

**if** exp1 **==** exp2:

**return** []

**elif** isConstant(exp1) **and** isConstant(exp2):

**if** exp1 **!=** exp2:

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

**return** []

**elif** isConstant(exp1):

**return** [(exp1, exp2)]

**elif** isConstant(exp2):

**return** [(exp2, exp1)]

**elif** isVariable(exp1):

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

**elif** isVariable(exp2):

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

**elif** getPredicate(exp1) **!=** getPredicate(exp2):

print(f"Predicates {getPredicate(exp1)} and {getPredicate(exp2)} do not match. Cannot be unified")

**return** []

**elif** len(getAttributes(exp1)) **!=** len(getAttributes(exp2)):

print(f"Length of attributes {len(getAttributes(exp1))} and {len(getAttributes(exp2))} do not match. Cannot be unified")

**return** []

*# Unify first attributes*

firstAttr1 **=** getFirstAttribute(exp1)

firstAttr2 **=** getFirstAttribute(exp2)

initialSubstitution **=** unify(firstAttr1, firstAttr2)

**if** **not** initialSubstitution:

**return** []

**if** len(getAttributes(exp1)) **==** 1:

**return** initialSubstitution

remainingAttr1 **=** getRemaining(exp1)

remainingAttr2 **=** getRemaining(exp2)

**if** initialSubstitution **!=** []:

*# Check if there's "nested" unification. eg. [x, y] and [y, z] in the other attributes*

remainingAttr1 **=** apply(remainingAttr1, initialSubstitution)

remainingAttr2 **=** apply(remainingAttr2, initialSubstitution)

*# Recursively unify other attributes*

remainingSubstitution **=** unify(remainingAttr1, remainingAttr2)

**if** **not** remainingSubstitution:

**return** []

**return** initialSubstitution **+** remainingSubstitution

exp1 **=** input("Expresssion 1: ")

*# Like(f(z),B)*

exp2 **=** input("Expresson 2: ")

*# Like(x,y)*

substitutions **=** unify(exp1, exp2)

print("Substitutions:")

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

exp1 **=** input("Expresssion 1: ")

*# Like(f(y),z)*

exp2 **=** input("Expresssion 2: ")

*# Like(M,f(y))*

substitutions **=** unify(exp1, exp2)

print("Substitutions:")

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

exp1 **=** input("Expresssion 1: ")

*# Like(A,z)*

exp2 **=** input("Expresssion 2: ")

*# Like(M,f(y))*

substitutions **=** unify(exp1, exp2)

print("Substitutions:")

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

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

**def** getAttributes(string):

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

matches **=** re**.**findall(expr, string)

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

**def** getPredicates(string):

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

**return** re**.**findall(expr, string)

**def** 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**:

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

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

**return** statement

**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

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

import re

def isVariable(x):

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

def getAttributes(string):

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

matches = re.findall(expr, string)

return matches

def getPredicates(string):

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

return re.findall(expr, string)

class Fact:

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

self.expression = expression

predicate, params = self.splitExpression(expression)

self.predicate = predicate

self.params = params

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

def splitExpression(self, expression):

predicate = getPredicates(expression)[0]

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

return [predicate, params]

def getResult(self):

return self.result

def getConstants(self):

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

def getVariables(self):

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

def substitute(self, constants):

c = constants.copy()

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

return Fact(f)

class Implication:

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

self.expression = expression

l = expression.split('=>')

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

self.rhs = Fact(l[1])

def evaluate(self, facts):

constants = {}

new\_lhs = []

for fact in facts:

for val in self.lhs:

if val.predicate == fact.predicate:

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

if v:

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

new\_lhs.append(fact)

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

for key in constants:

if constants[key]:

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

expr = f'{predicate}{attributes}'

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

class KB:

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

self.facts = set()

self.implications = set()

def tell(self, e):

if '=>' in e:

self.implications.add(Implication(e))

else:

self.facts.add(Fact(e))

for i in self.implications:

res = i.evaluate(self.facts)

if res:

self.facts.add(res)

def query(self, e):

facts = set([f.expression for f in self.facts])

i = 1

print(f'Querying {e}:')

for f in facts:

if Fact(f).predicate == Fact(e).predicate:

print(f'\t{i}. {f}')

i += 1

def display(self):

print("All facts: ")

for i, f in enumerate(set([f.expression for f in self.facts])):

print(f'\t{i+1}. {f}')

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()

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)')

kb.display()