

SRM INSTITUTE OF SCIENCE & TECHNOLOGY DEPARTMENT OF NETWORKING & COMMUNICATIONS **18CSC305J-ARTIFICIAL INTELLIGENCE**

SEMESTER – 6 BATCH-2

|  |  |
| --- | --- |
| **REGISTRATION NUMBER** | **RA1911003011043** |
| **NAME** | **RAMNATH** |

# B.Tech- CSE / CC, Third Year (Section: H2)

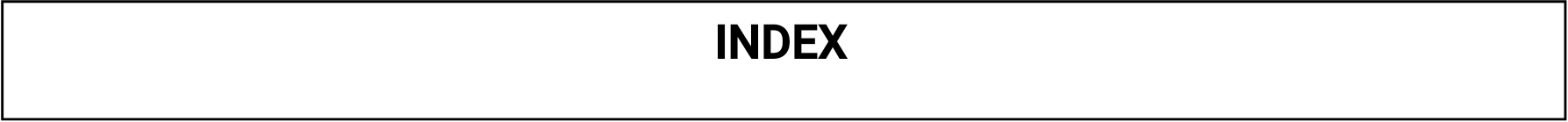
**Faculty In charge: Dr. S. Prabakeran, B.Tech, M.E, PhD**

**Assistant Professor**

**School of Computing - Department of**

**Networking and Communications**

**Year 2021-2022 / Even Semester**



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Ex No** | **DATE** | **Title** | **Page No** | **Marks** |
| 8 | 15/03/02 | Implementation of unification and resolution for real world problems. |  |  |

## Experiment No: 8

**Date : 15-03-22**

**UNIFICATION AND RESOLUTION**

**AIM:** Developing an algorithm for implementation of **UNIFICATION AND RESOLUTION**

**PROBLEM STATEMENT:** Developing an optimized technique using an appropriate artificial intelligence algorithm to solve the Unification and Resolution.

## ALGORITHM :

1. function PL-RESOLUTION (KB, Q) returns true or false inputs: KB,
2. the knowledge base, group of sentences/facts in propositional logic
3. Q, the query, a sentence in propositional logic
4. clauses → the set of clauses in the CNF representation of KB ^ Q new → { }
5. loop do for each Ci, Cj in clauses do
6. resolvents → PL-RESOLVE (Ci, Cj)
7. if resolvents contains the empty clause the return true
8. new → new union resolvents
9. if new is a subset of clauses then return false
10. clauses → clauses union true

## OPTIMIZATION TECHNIQUE:

Resolution basically works by using the principle of proof by contradiction. To find the conclusion we should negate the conclusion. Then the resolution rule is applied to the resulting clauses. Each clause that contains complementary literals is resolved to produce a2. new clause, which can be added to the set of facts (if it is not already present). This process continues until one of the two things happen:•There are no new clauses that can be added. An application of the resolution rule derives the empty clauseAn empty clause shows that the negation of the conclusion is a complete contradiction,hence the negation of the conclusion is invalid or false or the assertion is completely valid or true.

1. Convert the given statements in Predicate/Propositional Logic
2. Convert these statements into Conjunctive Normal Form
3. Negate the Conclusion (Proof by Contradiction)
4. Resolve using a Resolution Tree (Unification)

## UNIFICATION CODE:

def get\_index\_comma(string): index\_list = list() par\_count = 0

for i in range(len(string)):

if string[i] == ',' and par\_count == 0: index\_list.append(i)

elif string[i] == '(': par\_count += 1 elif string[i] == ')':

par\_count -= 1 return index\_list

def is\_variable(expr): for i in expr:

if i == '(' or i == ')': return False

return True

def process\_expression(expr): expr = expr.replace(' ', '') index = None

for i in range(len(expr)): if expr[i] == '(':

index = i break

predicate\_symbol = expr[:index]

expr = expr.replace(predicate\_symbol, '') expr = expr[1:len(expr) - 1]

arg\_list = list()

indices = get\_index\_comma(expr)

if len(indices) == 0: arg\_list.append(expr)

else:

arg\_list.append(expr[:indices[0]]) for i, j in zip(indices, indices[1:]): arg\_list.append(expr[i + 1:j])

arg\_list.append(expr[indices[len(indices) - 1] + 1:]) return predicate\_symbol, arg\_list

def get\_arg\_list(expr):

\_, arg\_list = process\_expression(expr)

flag = True while flag:

flag = False

for i in arg\_list:

if not is\_variable(i): flag = True

\_, tmp = process\_expression(i) for j in tmp:

if j not in arg\_list: arg\_list.append(j)

arg\_list.remove(i) return arg\_list

def check\_occurs(var, expr): arg\_list = get\_arg\_list(expr) if var in arg\_list:

return True return False

def unify(expr1, expr2):

if is\_variable(expr1) and is\_variable(expr2): if expr1 == expr2:

return 'Null' else:

return False

elif is\_variable(expr1) and not is\_variable(expr2): if check\_occurs(expr1, expr2):

return False else:

tmp = str(expr2) + '/' + str(expr1) return tmp

elif not is\_variable(expr1) and is\_variable(expr2): if check\_occurs(expr2, expr1):

return False else:

tmp = str(expr1) + '/' + str(expr2) return tmp

else:

predicate\_symbol\_1, arg\_list\_1 = process\_expression(expr1) predicate\_symbol\_2, arg\_list\_2 = process\_expression(expr2)

# Step 2

if predicate\_symbol\_1 != predicate\_symbol\_2: return False

# Step 3

elif len(arg\_list\_1) != len(arg\_list\_2): return False

else:

# Step 4: Create substitution list sub\_list = list()

# Step 5:

for i in range(len(arg\_list\_1)):

tmp = unify(arg\_list\_1[i], arg\_list\_2[i])

if not tmp: return False

elif tmp == 'Null': pass

else:

if type(tmp) == list: for j in tmp:

sub\_list.append(j)

else:

sub\_list.append(tmp)

# Step 6 return sub\_list

if name == ' main ': f1 = 'Q(a, g(x, a), f(y))'

f2 = 'Q(a, g(f(b), a), x)' # f1 = input('f1 : ')

# f2 = input('f2 : ')

result = unify(f1, f2) if not result:

print('The process of Unification failed!') else:

print('The process of Unification successful!') print(result)

## RESOLUTION CODE:

import copy import time

class Parameter: variable\_count = 1

def init (self, name=None): if name:

self.type = "Constant" self.name = name

else:

self.type = "Variable"

self.name = "v" + str(Parameter.variable\_count)

Parameter.variable\_count += 1

def isConstant(self):

return self.type == "Constant"

def unify(self, type\_, name): self.type = type\_ self.name = name

def eq (self, other):

return self.name == other.name

def str (self): return self.name

class Predicate:

def init (self, name, params): self.name = name

self.params = params

def eq (self, other):

return self.name == other.name and all(a == b for a, b in zip(self.params, other.params))

def str (self):

return self.name + "(" + ",".join(str(x) for x in self.params) + ")"

def getNegatedPredicate(self):

return Predicate(negatePredicate(self.name), self.params)

class Sentence: sentence\_count = 0

def init (self, string):

self.sentence\_index = Sentence.sentence\_count Sentence.sentence\_count += 1

self.predicates = [] self.variable\_map = {} local = {}

for predicate in string.split("|"):

name = predicate[:predicate.find("(")] params = []

for param in predicate[predicate.find("(") + 1: predicate.find(")")].split(","):

if param[0].islower():

if param not in local: # Variable local[param] = Parameter()

self.variable\_map[local[param].name] = local[param] new\_param = local[param]

else:

new\_param = Parameter(param) self.variable\_map[param] = new\_param

params.append(new\_param) self.predicates.append(Predicate(name, params))

def getPredicates(self):

return [predicate.name for predicate in self.predicates]

def findPredicates(self, name):

return [predicate for predicate in self.predicates if predicate.name == name]

def removePredicate(self, predicate): self.predicates.remove(predicate)

for key, val in self.variable\_map.items(): if not val:

self.variable\_map.pop(key)

def containsVariable(self):

return any(not param.isConstant() for param in self.variable\_map.values())

def eq (self, other):

if len(self.predicates) == 1 and self.predicates[0] == other: return True

return False

def str (self):

return "".join([str(predicate) for predicate in self.predicates])

class KB:

def init (self, inputSentences):

self.inputSentences = [x.replace(" ", "") for x in inputSentences] self.sentences = []

self.sentence\_map = {}

def prepareKB(self): self.convertSentencesToCNF()

for sentence\_string in self.inputSentences: sentence = Sentence(sentence\_string) for predicate in sentence.getPredicates():

self.sentence\_map[predicate] = self.sentence\_map.get( predicate, []) + [sentence]

def convertSentencesToCNF(self):

for sentenceIdx in range(len(self.inputSentences)):

# Do negation of the Premise and add them as literal if "=>" in self.inputSentences[sentenceIdx]:

self.inputSentences[sentenceIdx] = negateAntecedent( self.inputSentences[sentenceIdx])

def askQueries(self, queryList): results = []

for query in queryList:

negatedQuery = Sentence(negatePredicate(query.replace(" ", ""))) negatedPredicate = negatedQuery.predicates[0] prev\_sentence\_map = copy.deepcopy(self.sentence\_map)

self.sentence\_map[negatedPredicate.name] = self.sentence\_map.get( negatedPredicate.name, []) + [negatedQuery]

self.timeLimit = time.time() + 40

try:

result = self.resolve([negatedPredicate], [

False]\*(len(self.inputSentences) + 1))

except:

result = False

self.sentence\_map = prev\_sentence\_map if result:

results.append("TRUE") else:

results.append("FALSE") return results

def resolve(self, queryStack, visited, depth=0): if time.time() > self.timeLimit:

raise Exception if queryStack:

query = queryStack.pop(-1)

negatedQuery = query.getNegatedPredicate() queryPredicateName = negatedQuery.name

if queryPredicateName not in self.sentence\_map: return False

else:

queryPredicate = negatedQuery

for kb\_sentence in self.sentence\_map[queryPredicateName]: if not visited[kb\_sentence.sentence\_index]:

for kbPredicate in kb\_sentence.findPredicates(queryPredicateName): canUnify, substitution = performUnification(

copy.deepcopy(queryPredicate), copy.deepcopy(kbPredicate))

if canUnify:

newSentence = copy.deepcopy(kb\_sentence) newSentence.removePredicate(kbPredicate) newQueryStack = copy.deepcopy(queryStack)

if substitution:

for old, new in substitution.items():

if old in newSentence.variable\_map:

parameter = newSentence.variable\_map[old] newSentence.variable\_map.pop(old) parameter.unify(

"Variable" if new[0].islower() else "Constant", new) newSentence.variable\_map[new] = parameter

for predicate in newQueryStack:

for index, param in enumerate(predicate.params): if param.name in substitution:

new = substitution[param.name] predicate.params[index].unify(

"Variable" if new[0].islower() else "Constant", new)

for predicate in newSentence.predicates: newQueryStack.append(predicate)

new\_visited = copy.deepcopy(visited) if kb\_sentence.containsVariable() and

len(kb\_sentence.predicates) > 1:

new\_visited[kb\_sentence.sentence\_index] = True

if self.resolve(newQueryStack, new\_visited, depth + 1): return True

return False return True

def performUnification(queryPredicate, kbPredicate): substitution = {}

if queryPredicate == kbPredicate: return True, {}

else:

for query, kb in zip(queryPredicate.params, kbPredicate.params): if query == kb:

continue

if kb.isConstant():

if not query.isConstant():

if query.name not in substitution: substitution[query.name] = kb.name

elif substitution[query.name] != kb.name:

return False, {} query.unify("Constant", kb.name)

else:

return False, {}

else:

if not query.isConstant():

if kb.name not in substitution: substitution[kb.name] = query.name

elif substitution[kb.name] != query.name: return False, {}

kb.unify("Variable", query.name) else:

if kb.name not in substitution: substitution[kb.name] = query.name

elif substitution[kb.name] != query.name: return False, {}

return True, substitution

def negatePredicate(predicate):

return predicate[1:] if predicate[0] == "~" else "~" + predicate

def negateAntecedent(sentence):

antecedent = sentence[:sentence.find("=>")] premise = []

for predicate in antecedent.split("&"): premise.append(negatePredicate(predicate))

premise.append(sentence[sentence.find("=>") + 2:]) return "|".join(premise)

def getInput(filename):

with open(filename, "r") as file: noOfQueries = int(file.readline().strip())

inputQueries = [file.readline().strip() for \_ in range(noOfQueries)] noOfSentences = int(file.readline().strip())

inputSentences = [file.readline().strip() for \_ in range(noOfSentences)]

return inputQueries, inputSentences

def printOutput(filename, results): print(results)

with open(filename, "w") as file: for line in results:

file.write(line) file.write("\n")

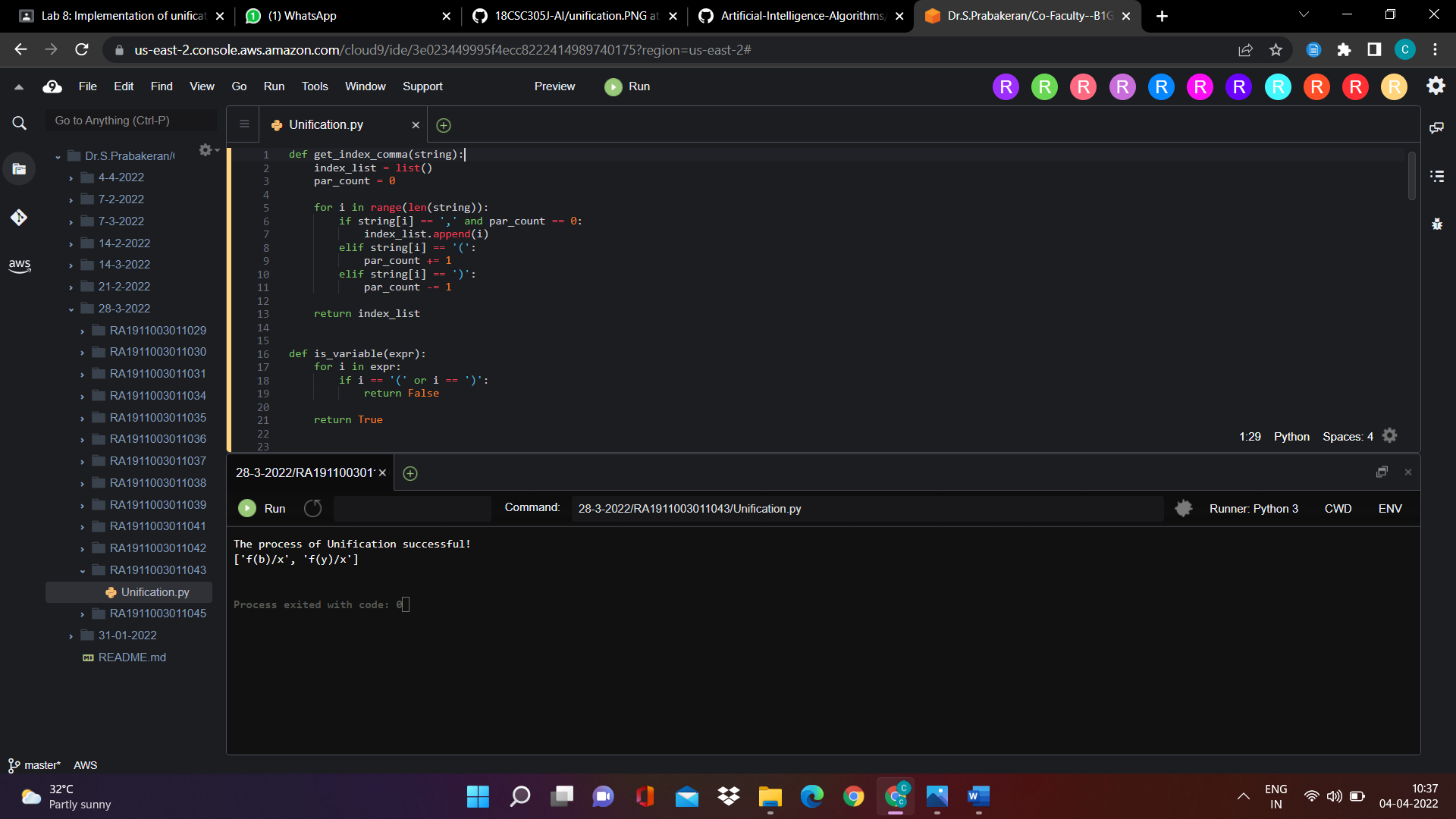
file.close()

if name == ' main ':

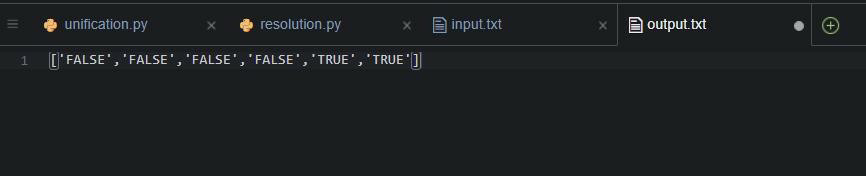
inputQueries\_, inputSentences\_ = getInput('input.txt') knowledgeBase = KB(inputSentences\_) knowledgeBase.prepareKB()

results\_ = knowledgeBase.askQueries(inputQueries\_) printOutput("output.txt", results\_)

# UNIFICATION OUTPUT:



RESOLUTION OUTPUT:



# RESULT:Developed Unification and Resolution Algorithm in Python for solving logical problems.