1103b2

Experiment-7

Implementation of resolution for real-world problems

Aim: To implement resolution for real-world problems

Algorithm:
1) function Resolution(kb, α) returns true/false
2) knowledge base, group of facts in propositional logic ← kb
3) α - query, sentence in propositional logic
4) clauses - set of clauses in CNF representation of kb^α new → {}
5) loop do for each C_i, C_j in clauses do
6) resolvents → Resolve(C_i, C_j)
7) if resolvents contains the empty clause then return true
8) new → new union resolvents
9) if new is a subset of clauses then return false
10) clauses → clauses union true

Result: Resolution has been successfully implemented for real world problems.

Dhawal Patil
RA1911003010575
CSE A2

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()

inputQueries_, inputSentences_ = getInput('untitled2.txt')
knowledgeBase = KB(inputSentences_)
knowledgeBase.prepareKB()
results_ = knowledgeBase.askQueries(inputQueries_)
printOutput("output.txt", results_)
```

Screenshot

Input

Input 1 -

```
3
AtRisk(Bob)
LivesWith(Alice,Bob)
Take(Bob,VitC)
9
HighSugar(x) & HighBP(x) => AtRisk(x)
AtRisk(x) & LivesWith(x,y) => AtRisk(y)
Take(Alice,x) => Take(Bob,x)
Take(Bob,x) => Take(Alice,x)
HighSugar(Alice)
HighBP(Bob)
HighSugar(Bob)
Take(Alice,VitC)
~AtRisk(Alice)
```

Input 2 -

```
2
CallAmbulance(Alice)
CallAmbulance(Bob)
15
HasSymptom(x,a) & SOS(a) => CallAmbulance(x)
HasSymptom(x,a) & HasSymptom(x,b) => HasTwoSymptoms(x,a,b)
HasTwoSymptoms(x,Pain,y) & ~SOS(y) => Take(x,Tylenox)
HasTwoSymptoms(x,Pain,y) & SOS(y) => CallAmbulance(x)
HasSymptom(x,Cough) => Take(x,CoughDrop)
HasSymptom(x,Pain) => Take(x,Tylenox)
HasTwoSymptoms(x,HighBP,Pregnant) => ~Take(x,DragonClaw)
HasFiveSymptoms(x,Dehydrated,Headache,HighBP,HighFever,Pain) => CallAmbulance(x)
SOS(HeartAttack)
SOS(Stroke)
~SOS(Cold)
~SOS(Cough)
HasSymptom(Alice,Pain)
HasSymptom(Alice,Stroke)
HasFiveSymptoms(Bob,Cough,Sleepy,Vomit,HighFever,Pain)
```

Output

Output 1 -

```
['TRUE', 'FALSE', 'TRUE']
```

Output 2 -

```
['TRUE', 'FALSE']
```