人工智能实验报告

课程名称：**Artificial Intelligence**

专业（方向）：信息与计算科学

学号：23323035

姓名：崔行健

实验题目：归结原理实验

---

1. 实验内容

1.1 算法原理

一阶逻辑归结算法的核心是通过消解互补文字对推导出空子句，从而证明查询的否定与知识库矛盾。具体步骤如下：

1. **子句标准化**：将知识库和查询转换为析取式，统一变量命名避免冲突。
2. **寻找互补对**：遍历子句对，若存在文字`L1`和`~L2`（或反之）且可通过合一（Unification）匹配，则进行消解。
3. **最一般合一（MGU）**：为互补文字对中的变量和常量建立替换规则，使两者结构一致。
4. **生成新子句**：应用MGU替换后，合并两个子句并删除互补对，生成新子句。若新子句为空，则证明完成。
5. **支持集策略**：
   - 每次归结时，两个亲本子句中至少要有一个是目标公式否定的子句或其后裔。
   - 支持集 = 目标公式否定的子句集合 ∪ 这些子句通过归结生成的所有后裔子句

   特点：

   - 尽量避免在可满足的子句集中做归结，因为从中导不出空子句。而求证公式的前提通常是一致的，所以支持集策略要求归结时从目标公式否定的子句出发进行归结。支持集策略实际是一种目标制导的反向推理。
   - 支持集策略是完备的。

1.2 伪代码

MGU算法伪代码

```
function MGU(args1, args2):
    if len(args1) != len(args2): return None
    unification = {}
    for i in 0 to len(args1)-1:
        val1 = args1[i], val2 = args2[i]
        if val1 == val2: continue
        if val1是变量且val2是常量:
            unification[val1] = val2
        elif val2是变量且val1是常量:
            unification[val2] = val1
        else: return None  # 无法合一
    return unification
```

归结算法伪代码

```
function resolution(KB, query):
    clauses = KB + [query]
    while True:
        生成所有可能的子句对(C1, C2)
        for each (C1, C2) in clauses:
            for L1 in C1, L2 in C2:
                if L1和L2是互补对:
                    mgu = MGU(L1, L2)
                    if mgu存在:
                        应用mgu到C1和C2，生成新子句C_new
                        if C_new为空: 返回"证明成功"
                        if C_new不在clauses中: 添加到clauses
        if 无新子句生成: 返回"证明失败"
```

1.3 关键代码展示

互补判断

```
def is_complement(self, literal1, literal2):
    pred1 = literal1.split('(')[0].lstrip('~')
    pred2 = literal2.split('(')[0].lstrip('~')
    return (pred1 == pred2) and (
        (literal1.startswith('~') ^ literal2.startswith('~'))
    )
```

MGU实现

```
def mgu(self, literal1, literal2):
    args1 = self.get_arguments(literal1)
    args2 = self.get_arguments(literal2)
    if len(args1) != len(args2): return None
    unification = {}
    for val1, val2 in zip(args1, args2):
        if val1 == val2: continue
        if self.is_variable(val1) and self.is_constant(val2):
            unification[val1] = val2
        elif self.is_variable(val2) and self.is_constant(val1):
            unification[val2] = val1
        else: return None
    return unification
```

归结步骤生成

```
def resolve(self, clause1, clause2, lit1_idx, lit2_idx):
    new_clause = list(clause1[:lit1_idx] + clause1[lit1_idx+1:] +
                      clause2[:lit2_idx] + clause2[lit2_idx+1:])
    return tuple(OrderedDict.fromkeys(new_clause))  # 去重
```

主归结循环

```
from collections import OrderedDict

DEBUG = 0       # 调试模式开关

class Sentences:
    def __init__(self, path):
        self.clauses = []       # 存储所有子句
        self.step = []          # 存储归结步骤记录
        with open(path, 'r') as f:
            lines = [line.strip() for line in f]

        for line in lines:
            # 跳过空行和标题行
            if not line or line.lower() == "kb:" or line.lower() == "query:":
                continue

            # 分隔文字
            if line.startswith('(') and line.endswith(')'):
                line = line[1: -1]
            line = line[:-1]
            literals = []

            for lit in line.split("),"):
                literals.append(lit.replace(" ", "") + ")")
            self.clauses.append(tuple(literals))

        if DEBUG:   # 调试输出
            for item in self.clauses:
                print(item)
            print(self.clauses)

    """检查两个文字是否为互补对"""
    def is_complement(self, literal1, literal2):
        # if DEBUG:
        #     print("literal1 and literal2: ", type(literal1), type(literal2))
        end1 = literal1.find('(')
        end2 = literal2.find('(')
        if literal1.startswith('~') and literal1[1:end1] == literal2[:end2]:
            return True
        elif literal2.startswith('~') and literal1[:end1] == literal2[1:end2]:
            return True

        return False

    """判断是否为变量"""
    def is_variable(self, val):
        return isinstance(val, str) and val.islower() and len(val) == 1

    """判断是否为常量（小写多字母）"""
    def is_constant(self, val):
        return isinstance(val, str) and len(val) >= 2 and val.islower()

    """从文字中提取参数列表"""
    def get_arguments(self, literal):
        begin = literal.find('(')
        end = literal.find(')')
        return literal[begin + 1:end].split(',')

    """计算最一般合一"""
    def mgu(self, literal1, literal2):
        args1 = self.get_arguments(literal1)
        args2 = self.get_arguments(literal2)
        if len(args1) != len(args2):
            return False
        n = len(args1)
        unification = {}
        if DEBUG:
            print(args1, args2)

        """
        1. 如果都是变量，则不可合一
        2. 如果都是常量，且不相等，则不可合一
        3. 当一个是变量，一个是项（在这里案例中只有常量），可以合一
        """
        while True:
            if args1 == args2:
                return unification
            for i in range(n):
                val1 = args1[i]
                val2 = args2[i]
                if self.is_variable(val1) and self.is_variable(val2):
                    return None
                elif self.is_constant(val1) and self.is_constant(val2) and val1
!= val2:
                    return None
                elif self.is_variable(val1) and self.is_constant(val2):
                    unification[val1] = val2
                    # 应用替换到整个参数列表
                    args1 = [unification[val] if val in unification else val
for val in args1]
```

```
                    args2 = [unification[val] if val in unification else val for val in args2]
                elif self.is_constant(val1) and self.is_variable(val2):
                    unification[val2] = val1
                    args1 = [unification[val] if val in unification else val for val in args1]
                    args2 = [unification[val] if val in unification else val for val in args2]


    """应用合一替换到子句"""
    def substitute(self, unification, clause):
        newclause = []
        for literal in clause:
            args = self.get_arguments(literal)
            args = [unification[val] if val in unification else val for val in args]
            begin = literal.find('(')
            end = literal.find(')')
            newliteral = literal[:begin + 1] + ','.join(args) + literal[end:]
            newclause.append(newliteral)
        return tuple(newclause)

    """执行归结操作"""
    def resolve(self, clause1, clause2, literal1_index, literal2_index):
        newclause = list(clause1 + clause2)
        newclause.remove(clause1[literal1_index])
        newclause.remove(clause2[literal2_index])
        newclause = list(OrderedDict.fromkeys(newclause))

        return tuple(newclause)

    def index(self, literal_index,clause_index,length):
        if length == 1: #如果子句只有一个元素，则文字索引不再需要
            index = str(clause_index+1)
        else:           #否则将文字索引变为字母
            index = str(clause_index+1) + chr(ord('a')+literal_index)
        return index

    """生成文字索引标识"""
    def sequence(self, newclause,unification,index1,index2):
        string = ''
        if unification == {}:    #如果字典为空，说明不需要输出合一
            string += 'R[' + index1 + ',' + index2 + '] = '
        else:
            string += 'R[' + index1 + ',' + index2 + ']{'
            for key,value in unification.items():
                string += key + '=' + value + ','
            string = string[:-1]
            string += '} = '
        string += str(newclause)
        return string

    # 支持集策略下的归结推理
    def resolution(self):
        clauseset = self.clauses
        clauseset = list(OrderedDict.fromkeys(clauseset))       # 去重
        step = ['归结顺序:'] + self.clauses      #将0位置补充元素，确保编号和列表索引
对应
        supportset = [clauseset[-1]]
        while True:
            clauseset_len = len(clauseset)
            new_clauseset = []

            # 遍历子句集
            for clause1_index in range(clauseset_len):
                for clause2_index in range (clause1_index + 1, clauseset_len):
                    clause1 = clauseset[clause1_index]
                    clause2 = clauseset[clause2_index]
                    clause1_len = len(clause1)
                    clause2_len = len(clause2)
                    if clause1 not in supportset and clause2 not in supportset:
                        continue
                    for literal1_index in range(clause1_len):
                        for literal2_index in range(clause2_len):
                            literal1 = clause1[literal1_index]
                            literal2 = clause2[literal2_index]

                            # 判断是否为互补对
                            if self.is_complement(literal1, literal2):
                                if DEBUG:
                                    print(literal1, literal2, "is complement")

                                # 最一般合一项
                                unification = self.mgu(literal1, literal2)
                                if unification == None:
                                    break
                                if DEBUG:   # 未实现功能：若违法，则跳出循环
                                    if unification == False:
                                        print("谓词的参数个数必须相同")
                                        return False

                                # 最一般合一替换
                                newclause1 = self.substitute(unification,
clause1)
                                newclause2 = self.substitute(unification,
clause2)

                                # 归结
                                newclause =  self.resolve(newclause1,
newclause2, literal1_index, literal2_index)

                                # 检查是否为新子句
                                if newclause in clauseset or newclause in
new_clauseset:
                                    break
                                new_clauseset.append(newclause)

                                # 记录步骤
                                index1 = self.index(literal1_index,
clause1_index, clause1_len)
                                index2 = self.index(literal2_index,
clause2_index, clause2_len)
                                sequence = self.sequence(newclause,
unification, index1, index2)
```

```
                                step.append(sequence)

                                # 发现空子句则成功
                                if newclause == ():
                                    self.step = step
                                    return
                            literal2_index += 1
                        literal1_index += 1
            if new_clauseset:
                clauseset += new_clauseset
                supportset += new_clauseset
            else:
                return False

    #得到归结式的子句索引
    def Number(self, clause):
        start = clause.find('[')
        end = clause.find(']')
        number = clause[start+1:end].split(',')
        #将文字索引去掉
        num1 = int(''.join(item for item in number[0] if not item.isalpha()))
        num2 = int(''.join(item for item in number[1] if not item.isalpha()))
        return num1,num2

    #得到新归结式的子句索引
    def Renumber(self, num,result,useful_process,size):
        if num <= size: #如果是初始子句集的，直接返回
            return num
        #找到亲本子句
        sequence = result[num]
        begin = sequence.find('(')
        aim_clause = sequence[begin:]
        #找到亲本子句在化简子句集的编号
        for i in range(size+1,len(useful_process)):
            begin = useful_process[i].find('(')
            if useful_process[i][begin:] == aim_clause:
                return i

    #更新归结式
    def Resequence(self, sequence,num1,num2,newnum1,newnum2):
        # 第一次替换：替换第一个编号
        start = sequence.find(num1)
        end = start + len(num1)
        sequence = sequence[:start] + newnum1 + sequence[end:]
        # 第二次替换：替换第二个编号
        end = start + len(newnum1)
        start = sequence.find(num2, end)
        end = start + len(num2)
        sequence = sequence[:start] + newnum2 + sequence[end:]
        return sequence

    #化简归结过程
    def Simplify(self, result,size):
        base_process = result[0:size+1] #初始子句集
        useful_process = []             #有用子句集
        number = [len(result)-1]        #用作队列，先将空子句的索引入列
        while number != []:
            # print(number)
            number0 = number.pop(0)                 #提取队列首元素，即有用子句的索引
            useful_process.append(result[number0])  #将有用子句加入到有用子句集

            num1,num2 = self.Number(result[number0])     #得有用子句用到的亲本子句索引
            #如果是初始子句集就无需加入
            if num1 > size:
                number.append(num1)
            if num2 > size:
                number.append(num2)
        #得到新的归结过程
        useful_process.reverse()
        useful_process = base_process + useful_process
        #将归结过程重新编号
        for i in range(size+1,len(useful_process)):
            num1,num2 = self.Number(useful_process[i])
            newnum1 = str(self.Renumber(num1,result,useful_process,size))
            newnum2 = str(self.Renumber(num2,result,useful_process,size))
            useful_process[i] = self.Resequence(useful_process[i],str(num1),str(num2),newnum1,newnum2)
        return useful_process

    def reindex(self):
        if DEBUG:
            for item in self.step:
                print(item)
        new_result = self.Simplify(self.step,len(self.clauses))
        print(new_result[0])
        for i in range(1,len(new_result)):
            print(i,new_result[i])

if __name__ == "__main__":
    test1 = Sentences("test1.txt")
    test1.resolution()
    test1.reindex()
```

2. 实验结果及分析

测试案例运行结果

输入文件`test1.txt`内容：

```
KB:
A(tony)
A(mike)
A(john)
L(tony, rain)
L(tony, snow)
(~A(x), S(x), C(x))
(~C(y), ~L(y, rain))
(L(z, snow), ~S(z))
(~L(tony, u), ~L(mike, u))
(L(tony, v), L(mike, v))
QUERY:
(~A(w), ~C(w), S(w))
```

运行后输出:

```
归结顺序:
1 ('A(tony)',)
2 ('A(mike)',)
3 ('A(john)',)
4 ('L(tony,rain)',)
5 ('L(tony,snow)',)
6 ('~A(x)', 'S(x)', 'C(x)')
7 ('~C(y)', '~L(y,rain)')
8 ('L(z,snow)', '~S(z)')
9 ('~L(tony,u)', '~L(mike,u)')
10 ('L(tony,v)', 'L(mike,v)')
11 ('~A(w)', '~C(w)', 'S(w)')
12 R[1,11a]{w=tony} = ('~C(tony)', 'S(tony)')
13 R[2,11a]{w=mike} = ('~C(mike)', 'S(mike)')
14 R[6c,12a]{x=tony} = ('~A(tony)', 'S(tony)')
15 R[6c,13a]{x=mike} = ('~A(mike)', 'S(mike)')
16 R[8b,14b]{z=tony} = ('L(tony,snow)', '~A(tony)')
17 R[8b,15b]{z=mike} = ('L(mike,snow)', '~A(mike)')
18 R[9a,16a]{u=snow} = ('~L(mike,snow)', '~A(tony)')
19 R[2,17b] = ('L(mike,snow)',)
20 R[19,18a] = ('~A(tony)',)
21 R[1,20] = ()
```

输入文件`test2.txt`内容:

```
KB:
GradStudent(sue)
(~GradStudent(x), Student(x))
(~Student(x), HardWorker(x))
QUERY:
~HardWorker(sue)
```

运行后输出:

```
归结顺序:
1 ('GradStudent(sue)',)
2 ('~GradStudent(x)', 'Student(x)')
3 ('~Student(x)', 'HardWorker(x)')
4 ('~HardWorker(sue)',)
5 R[3b,4]{x=sue} = ('~Student(sue)',)
6 R[2b,5]{x=sue} = ('~GradStudent(sue)',)
7 R[1,6] = ()
```

输入文件`test3.txt`内容：

```
KB:
On(aa,bb)
On(bb,cc)
Green(aa)
~Green(cc)
QUERY:
(~On(x,y), ~Green(x), Green(y))
```

运行后输出：

```
归结顺序:
1 ('On(aa,bb)',)
2 ('On(bb,cc)',)
3 ('Green(aa)',)
4 ('~Green(cc)',)
5 ('~On(x,y)', '~Green(x)', 'Green(y)')
6 R[4,5c]{y=cc} = ('~On(x,cc)', '~Green(x)')
7 R[3,5b]{x=aa} = ('~On(aa,y)', 'Green(y)')
8 R[2,6a]{x=bb} = ('~Green(bb)',)
9 R[1,7a]{y=bb} = ('Green(bb)',)
10 R[9,8] = ()
```

结果分析

1. **正确性**：成功推导出空子句，证明原查询`~B(tony)`与知识库矛盾，验证算法正确性。
2. **步骤清晰性**：输出按步骤展示归结过程，符合参考文档的格式要求。
3. **局限性**：当前实现仅支持变量与常量的合一，未处理含函数项或多元谓词的复杂场景。

核心代码说明：完整代码见附件，关键方法已在上文展示。