中山大学计算机学院

人工智能

本科生实验报告

（2023 学年春季学期）

一、 实验题目

归结原理

二、 实验内容

1. 算法原理

一阶逻辑归结算法是一种基于逻辑的自动推理技术，用于确定一组一阶逻辑子句是否一致（即可能同时为真）或不一致（即不可能同时为真）。这种算法的核心是归结原则，它是一种用于从子句集中推导出新子句的规则。归结原则的核心思想是寻找两个互补的文字（即一个是肯定的，另一个是否定的），并消除它们，从而产生一个新的子句。

本程序代码的设计分为以下几个步骤：

1、利用 Python 的读文件操作进行测试用例的读取，并将读取结果保存起来，保存的结构为 `list[tuple[str]]` 。list 中保存所有的子句，对于每一个子句，采用 `tuple[str]` 的结构进行保存，元组内保存文字（谓词），结构为 str。

2、文字解析。将子句中的文字（谓词）传入函数进行解析，返回一个列表，列表中保存文字的信息，结构为 `[bool（否定词）, 谓词, 参数 1, 参数 2...]`

3、MGU 算法。对两个文字进行 MGU 算法，函数传入对文字解析得到的列表，返回 bool 值，为 True 表示可以替换，False 表示不可替换，同时对替换字典进行修改，字典中 key 保存变量，value 保存替换后的项，包括变量、常量和函数。

4、生成归结子句。应用 MGU 替换到原始子句中，生成一个新的归结子句。归结子句是通过消除互补文字对得到的，它不包含任何互补的文字。

5、循环以不断归结，直到生成空子句或不可再次归结，终止算法。

6、打印归结步骤。这里需要注意的是，在子句的归结过程中可能会产生无用的归结步骤，这时要避免对其进行打印。**我采用的解决思路是，直到归结产生空子句后才开始打印归结步骤，采取回溯的办法，如果 c 子句由 a 子句和 b 子句归结产生，将 c 子句看作二叉树的根节点，a 和 b 为左右孩子，只有 a 和 b 被打印，c 才会被打印，也就是采取二叉树的后序遍历进行递归的打印操作。**

2. 伪代码

这里仅给出关键函数的伪代码

1、解析列表

```
def parseLiteral(literal : str) -> list:
```

#解析文字，返回列表，[否定（bool），谓词，参数 1，参数 2...]，如果有否定词，那么第一个元素为 True

初始化一个空列表 res 用于存储解析结果。

检查 literal 字符串的第一个字符是否为否定符号 '~'。

如果是，将 res 列表的第一个元素设置为 True，然后从 literal 中移除第一个字符。

如果不是，将 res 列表的第一个元素设置为 False。

找到 literal 中第一个左括号 '(' 的位置 end。

提取 literal 中从开始到 end 位置的子字符串作为谓词，并将其添加到 res 列表中。

从 literal 中提取 end + 1 到最后的子字符串，并以 ',' 为分隔符将其分割为参数列表 parameters。

将 parameters 列表中的每个参数添加到 res 列表中。

返回填充好的 res 列表。

2、MGU 算法

```
def MGU(parsing_1 : list, parsing_2 : list, substitutions : dict = {}) -> bool:
```

#对两个文字解析完成的列表进行 MGU 算法，替换后修改字典中 key = 变量，value = 替换的项，返回 False 表示不可替换

检查两个解析后的列表 parsing_1 和 parsing_2 的谓词是否相同，以及它们的参数数量是否相等。如果不同，返回 False。

遍历 parsing_1 和 parsing_2 中的参数列表，比较每对参数。

如果两个参数相同，继续下一次循环。

如果参数是函数表达式（包含 '('），处理函数表达式，剥离出不同的部分。

如果两个参数都是常量，返回 False，因为常量不可替换。

如果一个参数是变量，执行 occur 检查，如果变量已经出现在另一个参数中，返回 False。

如果一个参数是变量，将其与另一个参数在 substitutions 字典中进行替换。

如果所有参数都可以成功地进行 MGU 替换，返回 True。

3. 关键代码展示（带注释）

产生并维护可归结列表

```
def getResolvableList(sentences : list[tuple[str]]) -> list[tuple]:
    #传入所有待归结的字句，返回可归结的列表，[字句对的编号，第几个谓词]
    resolvableList = []
    for i in range(len(sentences)):
        for j in range(i + 1, len(sentences)):
            m, n = isResolvable(sentences[i], sentences[j])
            if m != -1 and n != -1:
                resolvableList.append((i, j, m, n))
    return resolvableList


def updateResolvableList(sentences : list[tuple[str]], newSentence : tuple[str],
resolvableList : list[tuple]):
    #对于一个归结出的新子句，更新可归结列表，并且将其插入头部位置，新归结的子句优先继续
```

归结

```
    for i in range(len(sentences)):
        m, n = isResolvable(sentences[i], newSentence)
        if m != -1 and n != -1:
            resolvableList.append((i, len(sentences), m, n))
```

归结主函数

```
def resolve(sentences : list[tuple[str]], resolvableList : list[tuple]) :
    resolveRes = {}                                 #归结结果，key = 归结子句的编号，value
= 进行归结的母子句下标、文字下标、替换字典
    len0 = len(sentences)                           #原始长度
    while resolvableList != []:
        i, j, m, n = resolvableList[0]
        del resolvableList[0]
        sentence_1 = sentences[i]
        sentence_2 = sentences[j]
        parsing_1 = parseLiteral(sentence_1[m])
        parsing_2 = parseLiteral(sentence_2[n])
        substitutions = {}                          #key = 变量，value = 替换的量
        MGU(parsing_1, parsing_2, substitutions)
        newSentence = resolveSentence(sentence_1, sentence_2, m, n, substitutions)
        if newSentence in sentences or newSentence == ('1',):                   #去
重，并且去掉永真式
            continue

        resolveRes[len(sentences)] = (i, j, m, n, substitutions)
        updateResolvableList(sentences, newSentence, resolvableList)
        sentences.append(newSentence)
        #print(sentences)
        #print(resolvableList)
        if newSentence == ():
            break
    if sentences[-1] == ():
        usefulList = [x for x in range(len0)]
        get_usefulList(usefulList, len(sentences) - 1, resolveRes)
        usefulList.sort()
        index = 1
        helpPrint = {}                          #在有用归结列表中的下标
        for i in usefulList:
            helpPrint[i] = index
            index += 1
        for i in usefulList:
            if i < len0:
                print(sentences[i])
            else:
```

```
                re = ''
                if len(sentences[resolveRes[i][0]]) != 1 and
len(sentences[resolveRes[i][1]]) != 1:
                    re = str(helpPrint[resolveRes[i][0]]) + chr(resolveRes[i][2] +
ord('a')) + ',' + str(helpPrint[resolveRes[i][1]]) + chr(resolveRes[i][3] + ord('a'))
                if len(sentences[resolveRes[i][0]]) == 1 and
len(sentences[resolveRes[i][1]]) != 1:
                    re = str(helpPrint[resolveRes[i][0]]) + ',' +
str(helpPrint[resolveRes[i][1]]) + chr(resolveRes[i][3] + ord('a'))
                if len(sentences[resolveRes[i][0]]) != 1 and
len(sentences[resolveRes[i][1]]) == 1:
                    re = str(helpPrint[resolveRes[i][0]]) + chr(resolveRes[i][2] +
ord('a')) + ',' + str(helpPrint[resolveRes[i][1]])
                if len(sentences[resolveRes[i][0]]) == 1 and
len(sentences[resolveRes[i][1]]) == 1:
                    re = str(helpPrint[resolveRes[i][0]]) + ',' +
str(helpPrint[resolveRes[i][1]])
                #print(f'R[{re}]{resolveRes[i][4]}{sentences[i]}')
                print(f'R[{re}]', end = '')
                print('{', end = '')
                #for old, new in resolveRes[i][4].items():
                    #print(f'{old}={new}', end = ',')
                temp = list(resolveRes[i][4].keys())
                for j in range(len(temp)):
                    old = temp[j]
                    new = resolveRes[i][4][old]
                    if j < len(resolveRes[i][4]) - 1:
                        print(f'{old}={new}', end = ',')
                    else:
                        print(f'{old}={new}', end = '')
                print('}', end = '')
                print(sentences[i])

        print('-----------------------------------------')
    else:
        print('不可归结')
        print('-----------------------------------------')
```

4. 创新点&优化（如果有）

1、更加完善的 MGU 算法

依据人工智能教材给出的算法进行实现最一般合一，可以对变量、函数体内部变量进行替换，比如 `f(u)` 和 `f(g(x))` 执行替换，则可以正确的给出 `u = f(x)` 的置换，由此我们可以**正确的运行附加题 1 和附加题 2**。

2、优化归结算法性能

在寻找可归结语句时，相比于传统的利用 for 循环进行机械的一一比对，本算法**维护一张可归结列表**，在执行归结主函数前，首先遍历全部的子句集，生成一个可归结列表，每次需要进行归结时**仅需查询可归结列表即可**，归结出一个新子句后，更新可归结列表，利用这种办法，可以降低时间复杂度。设子句集的规模为 N，并且会随着归结逐渐增大，那么**传统的机械比对的复杂度为 $O(N^2)$**，而采用这种维护可归结列表的方式，**可以将时间复杂度降低为 O(N)**，这里的 N 为更新可归结列表的花费。

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

```
test for alpineclub.txt
('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)')
('~A(w)', '~C(w)', 'S(w)')
R[2,6a]{x=mike}('C(mike)', 'S(mike)')
R[2,11a]{w=mike}('S(mike)', '~C(mike)')
R[5,9a]{u=snow}('~L(mike,snow)',)
R[12a,13b]{}('S(mike)',)
R[8a,14]{z=mike}('~S(mike)',)
R[15,16]{}()
------------------------------------------------
test for blockworld.txt
('On(aa,bb)',)
('On(bb,cc)',)
('Green(aa)',)
('~Green(cc)',)
('~On(x,y)', '~Green(x)', 'Green(y)')
R[1,5a]{x=aa,y=bb}('Green(bb)', '~Green(aa)')
R[2,5a]{x=bb,y=cc}('Green(cc)', '~Green(bb)')
R[3,6b]{}('Green(bb)',)
R[4,7a]{}('~Green(bb)',)
R[8,9]{}()
------------------------------------------------
```

```
------------------------------------------------
test for additionalQuestion1.txt
('I(bb)',)
('U(aa,bb)',)
('~F(u)',)
('~I(y)', '~U(x,y)', 'F(f(z))')
('~I(v)', '~U(w,v)', 'E(w,f(w))')
R[1,4a]{y=bb}('F(f(z))', '~U(x,bb)')
R[2,6b]{x=aa}('F(f(z))',)
R[3,7]{u=f(z)}()
------------------------------------------------
test for additionalQuestion2.txt
('~P(aa)',)
('P(z)', '~Q(f(z),f(u))')
('Q(x,f(g(y)))', 'R(s)')
('~R(t)',)
R[1,2a]{z=aa}('~Q(f(aa),f(u))',)
R[3b,4]{s=t}('Q(x,f(g(y)))',)
R[5,6]{x=f(aa),u=g(y)}()
------------------------------------------------
```