第二次实验报告

实验2 归结原理

陳日康　信息与计算科学　22336049

一、实验要求

编写程序，实现一阶逻辑归结算法，并用于求解给出的两个逻辑推理问题。

要求输出格式如下（格式正确且过程正确即可）：

1. $(P(x), Q(g(x)))$
2. $(R(a), Q(z), \neg P(a))$
3. $R[1a, 2c]\ \{X = a\} \quad (Q(g(a)), R(a), Q(z))$

……

"R" 表示归结步骤

"1a" 表示第一个句子 (1-st) 中的第一个 (a-st) 个原子公式，即 $P(x)$。

"2c" 表示第二个句子 (2-ed) 中的第三个 (c-th) 个原子公式，即 $\neg P(a)$。

"1a" 和 "2c" 是冲突的，所以应用最小合一 $\{X = a\}$。

归结问题 1：Alpine Club

$A(tony)$
$A(mike)$
$A(john)$
$L(tony, rain)$
$L(tony, snow)$
$(\neg A(x), S(x), C(x))$
$(\neg C(y), \neg L(y, rain))$
$(L(z, snow), \neg S(z))$
$(\neg L(tony, u), \neg L(mike, u))$
$(L(tony, v), L(mike, v))$
$(\neg A(w), \neg C(w), S(w))$

归结问题 2：Block World

$On(aa, bb)$
$On(bb, cc)$
$Green(aa)$
$\neg Green(cc)$
$(\neg On(x, y), \neg Green(x), Green(y))$

二、算法原理

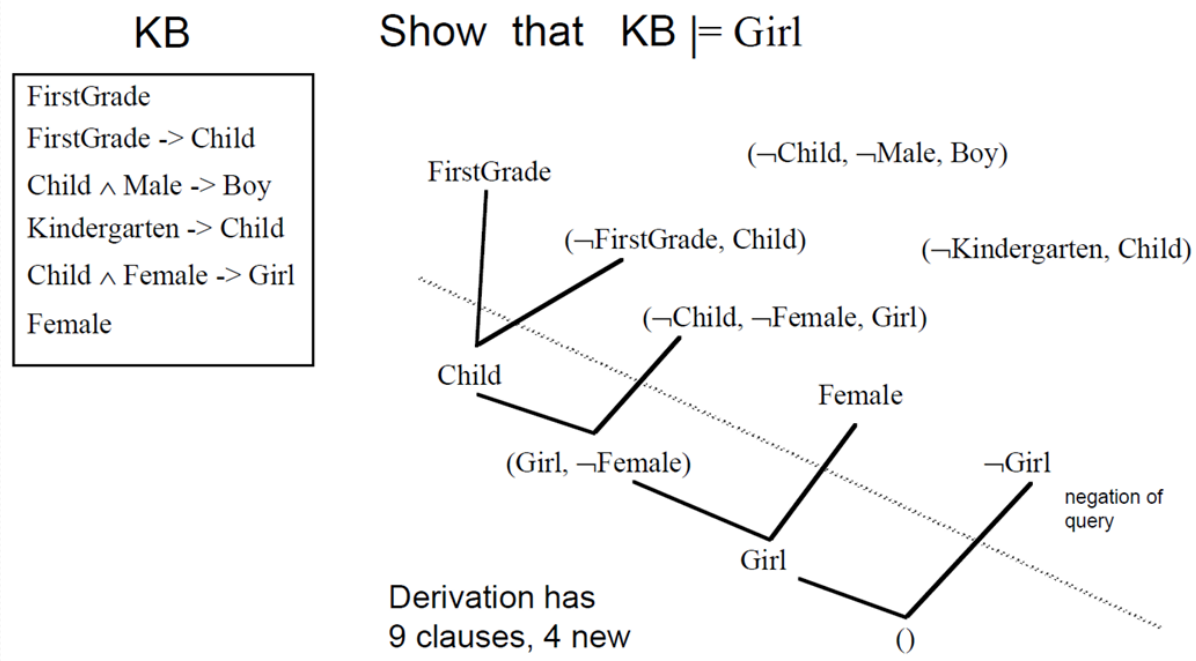
给定一组谓词逻辑子句，判断是否存在一种合一的方式使得这些子句归结为一个空子句，从而证明目标公式成立。归结是指对两个子句进行合一操作，消除它们的共同部分，从而生成一个新的子句。合一是指两个逻辑表达式之间的替换，使得它们可以在某种条件下统一为一个表达式。核心思想是采用宽度优先搜索策略进行穷举，直到最后消解到子句集为空时，从尾部开始向上找，寻找到有最短路径的解题思路（相当于建了个树）。

归结算法：

1. 首先将谓词逻辑子句转化为内部数据结构，并对每个子句进行编号。
2. 从初始子句集开始，采用宽度优先搜索的策略，两两逐一进行归结，生成第一级归结式集合。
3. 然后将初始子句集与第一级归结式集合进行归结，生成第二级归结式集合。
4. 依次类推，将前一级归结式集合与初始子句集、前一级和本级归结式集合进行归结，直到出现空子句或无法继续归结为止，算法结束。如果无法得到空子句，则目标公式不成立。

这个过程可以看作是在搜索一个归结树的过程，树的每一层代表一级归结式，从初始子句集出发，每一层通过归结生成下一级的归结式，直到最后找到空子句或无法继续归结。最终的解可以通过回溯树找到具有最短路径的解。

归结过程示例：



合一算法：

1. 如果两个符号是相等的常量或者变量，则它们是可合一的；否则是不可合一的。
2. 如果一个符号是变量，而另一个符号是常量或变量，则将变量绑定到常量或变量上，使它们相等。这个过程称为变量绑定。
3. 如果两个符号都是函数，且它们具有相同的函数名和相同数量的参数，则递归地比较它们的每个参数。如果每个参数都可合一，则两个函数可以合一；否则不可合一。

4. 如果两个符号都是复合符号，如一个带括号的逻辑表达式，那么将它们拆分成它们的子项，递归地应用上述步骤进行比较。

通过以上步骤，合一算法能够确定两个逻辑表达式之间的相等性。

三、伪代码

输入并处理子句集（包括删除多余的空号和空格）

创建答案列表

while 子句集不为空 {

- for i in range（0，当前训练集长度）
- 确定 j 长度：若当前为第一次循环，则更新为 i+1；否则更新为上一次最后的子句集长度
- for j in range（当前训练集长度）
- 检查 i 和 j 能否合并：
- 若能合并，则合并之后更新到答案集内，并将 i，j 一起合并到答案集中方便后续找到需要的项
- 不能合并就继续下一项
- 如果这一段合并的结果为空，则说明找到了正确答案，退出两重循环。

}

根据答案集中给出的 i，j，从尾部开始向上查找，并更新合并后的 i、j，插入新的答案集中

反向输出新的答案集，得到答案。

四、关键代码展示

`remove_space(s1)`是去除字符串中的多余空格。`remove_comma(l)`是去除字符串列表中逗号后面的空格。

```
def remove_space(s1):
    l = len(s1)
    i = 2
    while i < l:
        if s1[i] == " " and s1[i - 2] != ")":
            list1 = list(s1)
            list1.pop(i)
            s1 = ''.join(list1)
            l -= 1
        i += 1
    return s1
```

```
def remove_comma(l):
    for i in range(0,len(l)):
        if l[i][-1]==',':
            a_1=l[i][0:-1]
            l[i]=a_1
    return l
```

`get_opposite(s2)`：获取谓词的否定形式。如果输入字符串 `s2` 的第一个字符不是 "¬"，则在其前面添加 "¬"，否则去掉第一个字符。

```
def get_opposite(s2):
    ans_1 = ""
    if s2[0] != "¬":
        ans_1 = "¬" + s2
    else:
        ans_1 = s2[1:len(s2)]
    return ans_1
```

`weici(s3)` 是获取谓词的名称部分。`get_one_case(example_1)` 是获取单例。`get_first_part(example_2)` 是获取双例的前部分。`get_last_part(example_3)` 是获取双例的后部分。

```
def weici(s3):
    ans_2 = s3[0:s3.find("(")]
    return ans_2

def get_one_case(example_1):
    return example_1[example_1.find("(") + 1:example_1.find(")")]

def get_first_part(example_2):
    return example_2[example_2.find("(") + 1:example_2.find(",")]

def get_last_part(example_3):
    return example_3[example_3.find(",") + 1:example_3.find(")")]
```

`is_variable(f)` 是判断字符串是否为变量。`count_variables(statement)` 是统计变量的数量。

```
def is_variable(f):
    if f in ['x', 'y', 'z', 'u', 'v', 'w']:
        return True
    else:
        return False

def count_variables(statement):
    sum = 0
    for i in range(0, len(statement)):
        if is_variable(statement[i]) == True:
            sum += 1
    return sum
```

`judge_case(statement1, statement2)` 是判断两个单例或双例是否可合一。并返回合一结果。

```
def judge_case(statement1, statement2):
    ans = "none"
    if is_variable(statement1[0]):
        if statement1[1] != statement2[1]:
            return ""
        else:
            ans = "(" + statement1[0] + "=" + statement2[0] + ")"
            return ans
    elif is_variable(statement2[0]):
        if statement1[1] != statement2[1]:
            return ""
        else:
            ans = ans = "(" + statement2[0] + "=" + statement1[0] + ")"
            return ans
    elif is_variable(statement1[1]):
        if statement1[0] != statement2[0]:
            return ""
        else:
            ans = ans = "(" + statement1[1] + "=" + statement2[1] + ")"
            return ans
    elif is_variable(statement2[1]):
        if statement1[0] != statement2[0]:
            return ""
        else:
            ans = ans = "(" + statement2[1] + "=" + statement1[1] + ")"
            return ans
    elif statement1[0] == statement2[0] and statement1[1] == statement2[1]:
        return ans
    else:
        return ""
```

`judge(list1,list2)`：判断两个结点是否可合一。遍历两个结点的谓词列表，找到可以合一的谓词，并返回合一结果。

```
def judge(list1,list2):
```

```
    list1_index=-1
    list2_index=-1
    flag1=0
    for i in range(0,len(list1[0])):
        head_1=weici(list1[0][i])
        for j in range(0,len(list2[0])):
            head_2=weici(list2[0][j])
            if head_1==get_opposite(head_2):
                list1_index=i
                list2_index=j
                flag1=1
                break
        if flag1==1:
            break
    if list1_index==-1 and list2_index==-1: return []
    else:
        f1=str(list1[0][list1_index])
        f2=str(list2[0][list2_index])
        length=f1.count(",")
        if length==0:
            case1=get_one_case(f1)
            case2=get_one_case(f2)
            if case1==case2:
                return [list1_index,list2_index,"none"]
            elif is_variable(case1)==True and is_variable(case2)==False:
                return [list1_index,list2_index,"("+case1+"="+case2+")"]
            elif is_variable(case1)==False and is_variable(case2)==True:
                return [list1_index,list2_index,"("+case2+"="+case1+")"]
            else:
                return []
        elif length==1:
            case1=[get_first_part(f1),get_last_part(f1)]
            if count_variables(case1)>1: return []
            case2=[get_first_part(f2),get_last_part(f2)]
            if count_variables(case2)>1: return []
            x=judge_case(case1,case2)
            if x=="":return []
            else: return[list1_index,list2_index,x]
```

`unify(node_1, node_2, i, j, l)`：对两个结点进行合一操作。根据合一结果，更新结点列表 `l`。

```
def unify(node_1, node_2, i, j, l):
    re = judge(node_1, node_2)
    if len(re) == 0:
        return l
    l.append([])
    tail1 = dic[re[0]]
    tail2 = dic[re[1]]
    if len(node_1[0]) > 1:
        a = str(i) + str(tail1)
    else:
        a = str(i)
    if len(node_2[0]) > 1:
```

```
            b = str(j) + str(tail2)
        else:
            b = str(j)
        ss = re[2]
        t = []
        l1 = node_1[0][:]
        l2 = node_2[0][:]
        for m in range(0, len(l1)):
            l1[m] = l1[m].replace(ss[1], ss[3:len(ss) - 1])
        for m in range(0, len(l2)):
            l2[m] = l2[m].replace(ss[1], ss[3:len(ss) - 1])
        node_1[0] = l1
        node_2[0] = l2
        for n in range(0, len(node_1[0])):
            if n != re[0]:
                t.append(node_1[0][n])
        for n in range(0, len(node_2[0])):
            if n != re[1] and t.count(node_2[0][n]) == 0:
                t.append(node_2[0][n])
        l[len(l) - 1].append(t)
        l[len(l) - 1].append(a)
        l[len(l) - 1].append(b)
        l[len(l) - 1].append(ss)
        return l
```

`get_num(s4)` 是从谓词中获取变量编号。如果变量的编号是字母（`a` 到 `z`），则返回谓词的全部内容去除最后一个字符（即去除变量）。否则，返回原始谓词。而 `increment_variable_num(s5, num_1)` 是用于增加变量编号。它接受两个参数，`s5` 是谓词，`num_1` 是当前编号。如果谓词中的最后一个字符是字母，则将数字增加 1，并保留该字母作为新的编号的一部分。否则，只增加数字部分。

```
def get_num(s4):
    if s4[-1] <= "z" and s4[-1] >= "a":
        ans_2 = s4[0:-1]
    else:
        ans_2 = s4
    return ans_2

def increment_variable_num(s5, num_1):
    if s5[-1] <= "z" and s5[-1] >= "a":
        ans_3 = str(int(num_1) + 1) + s5[-1]
    else:
        ans_3 = str(int(num_1) + 1)
    return ans_3
```

`prepare_ans(l, node, ans, n)` 是准备输出结果。从合一后的结点列表中提取结果。对最后给出的答案集进行筛选（从尾部开始筛）当筛查到了最初的训练集时则停止递归。

```
def prepare_ans(l, node, ans, n):
    front = get_num(node[1])
    behind = get_num(node[2])
```

```
    if int(front) == -1 or int(behind) == -1:
        return
    tail = ""
    if node[len(node) - 1] == "none":
        tail = ""
    else:
        tail = node[3]
    f = front
    front = increment_variable_num(node[1], f)
    be = behind
    behind = increment_variable_num(node[2], be)
    # t=str("R["+str(front)+","+str(behind)+"]"+tail+" = "+str(node[0]))
    t = [str(front), str(behind), tail, str(node[0])]
    ans.append(t)
    if int(f) >= n: prepare_ans(l, l[int(f)], ans, n)
    if int(be) >= n: prepare_ans(l, l[int(be)], ans, n)
```

`correct_row(l, ans_d, s6)`：函数接受三个参数，`l`是包含所有谓词的列表，`ans_d`是包含解析结果的列表，`s6`是原始谓词的列表。首先，它将解析结果`ans_d`中的谓词追加到原始谓词列表`s6`中，形成完整的谓词集合。但是这样做会导致谓词之间的行数错误。接着，它遍历完整的谓词集合`s6`，从最后一个谓词向前遍历到第`n`个谓词（`n`是原始谓词数量），并且对谓词中的行数进行修正。修正的方式是，如果谓词中的行数大于`n`，则查找该行谓词对应的原始谓词在列表`l`中的位置，并用修正后的行数替换谓词中的行数。最后，它将修正后的解析结果更新到原始谓词列表`s6`中。

```
def correct_row(l, ans_d, s6):
    n = len(s6)
    for i in range(0, len(ans_d)):
        s6.append(ans_d[i])
    for i in range(len(s6) - 1, n - 1, -1):
        s_1 = int(get_num(s6[i][0])) - 1
        s_2 = int(get_num(s6[i][1])) - 1
        if s_1 >= n:
            ss1 = str(l[s_1][0])
            for j in range(n, len(s6)):
                if ss1 == s6[j][3]:
                    s6[i][0] = s6[i][0].replace(get_num(s6[i][0]), str(j + 1))
                    break
        if s_2 >= n:
            ss2 = str(l[s_2][0])
            for j in range(n, len(s6)):
                if ss2 == s6[j][3]:
                    s6[i][1] = s6[i][1].replace(get_num(s6[i][1]), str(j + 1))
                    break
    for i in range(0, len(ans_d)):
        ans_d[i] = s6[i + n]
```

`final(list_f, s))`实现了逻辑推理中的归结算法，两个参数分别是`list_f`是包含所有谓词的列表，`s`是原始谓词的列表。在函数内部，通过一个循环来执行归结算法，直到找到归结结果或者无法再进行推理。在循环内部，对于列表中的每一对谓词，都尝试进行归结操作。首先，它调用`unify()`函数来尝试找到可合一的谓词，如果找到了合一的谓词，就进行合一操作，将合一后的结果更新到列表中。然

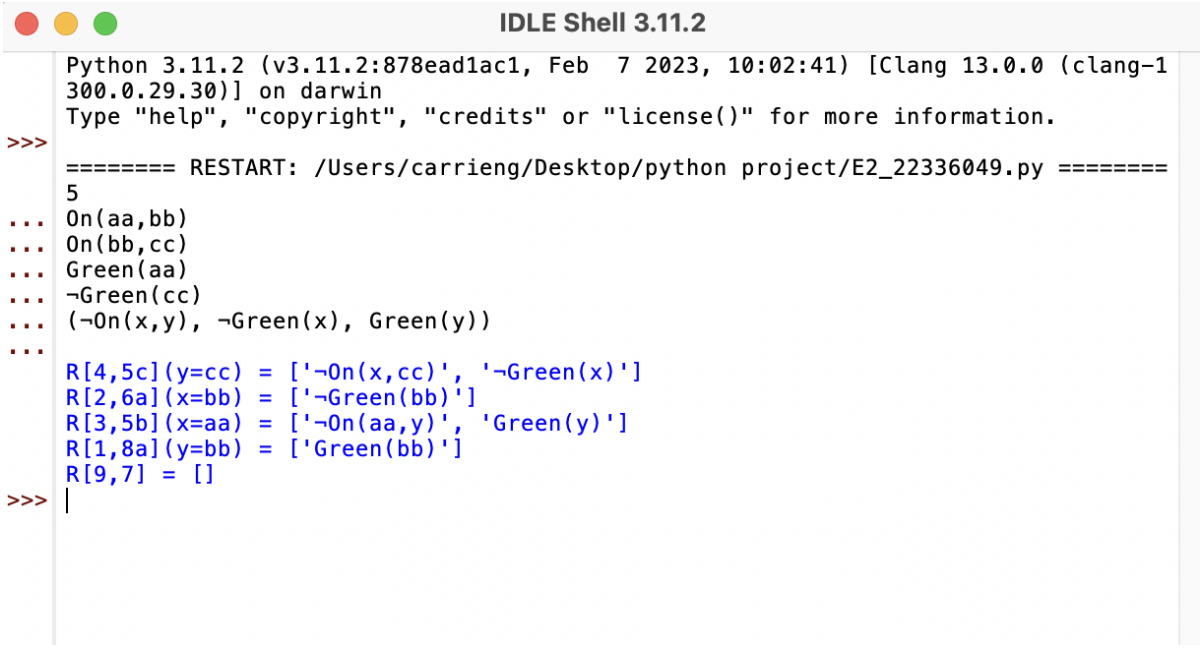
后，检查是否有一个空的谓词出现，如果有，说明归结成功，设置 `flag` 标志为 1 并跳出循环。如果在一轮循环中未找到归结结果，则增加 `sum` 变量的值，继续下一轮循环。在下一轮循环中，只遍历上一轮循环中找到的谓词对之后的谓词对，避免了重复计算。循环结束后，将归结结果反转并进行行数修正，最后打印输出。

```
def final(list_f, s):
    flag = 0
    sum = 0
    while True:
        l = len(list_f)
        for i in range(0, l):
            if sum == 0:
                j = i + 1
            else:
                j = l1
            for j in range(j, l):
                list1 = list_f[i]
                list2 = list_f[j]
                list_f = unify(list1[:], list2[:], i, j, list_f)
                if len(list_f[len(list_f) - 1][0]) == 0:
                    flag = 1
                    break
            if flag == 1:
                break

        if flag == 1:
            break
        sum += 1
        l1 = l
    ans = []
    prepare_ans(list_f, list_f[len(list_f) - 1], ans, np)
    ans.reverse()
    correct_row(list_f, ans, s)
    print_ans(ans)
```

五、实验结果

归结问题 1 的运行结果，结果正确。



归结问题 2 的运行结果，结果正确。

六、参考资料

https://blog.csdn.net/Zhangguohao666/article/details/105571115

https://blog.csdn.net/m0_51663233/article/details/134093476

https://blog.csdn.net/modeala_/article/details/17712275