中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	21311274	姓名	林宇浩

一、 实验题目

实现一阶逻辑归结算法

二、 实验内容

1. 算法原理

归结反演：

1、将前提条件中的合式公式转换为子句形式，本实验中前提条件数据已是子句的形式

2、将待验证的结论的否定也转换为子句形式，并添加到子句集中，本实验结论的否定已存在于前提条件中

3、反复对子句集中的子句应用归结规则，并把得到的归结式也加入到子句集中，如果产生空子句说明前提条件可以推出结论，如果直到再没有子句可以进行归结都没有产生空子句，说明无法退出结论

用到的搜索策略：

1、深度优先：

设前提条件和结论的否定为第 0 层归结式，即原始子句。进行深度优先搜索策略时，先用第 0 层归结式生成第 1 层归结式，再用第 1 层归结式和第 0 层归结式进行归结，得到第 2 层归结式，再用第 2 层归结式和第 0、1 层归结式归结，得到第 3 层归结式，重复上述步骤直至产生空子句为止，否则继续用新产生的层和其以下的层进行归结。

2、支持集策略：

首先先定义后裔的概念，若子句 b 是子句 a 的后裔，则 a 有参与归结推得 b 的过程。比如假设子句 a1、a2 归结得到 b1，子句 a3、a4 归结得到 b2，子句 b1、b2 归结得到 c1，则 b1 为 a1、a2 的后裔，b2 为 a3、a4 的后裔，c1 是 a1、a2、a3、a4、b1、b2 的后裔。采用支持集策略时，规定：每次归结时，参与归结的子句中必须至少有一个是结论的否定或结论的否定的后裔。

合一算法:

令 k 为计数器，W 为待合一集合，δ 为置换。

1、置 k=0，Wk=W，δ=6。

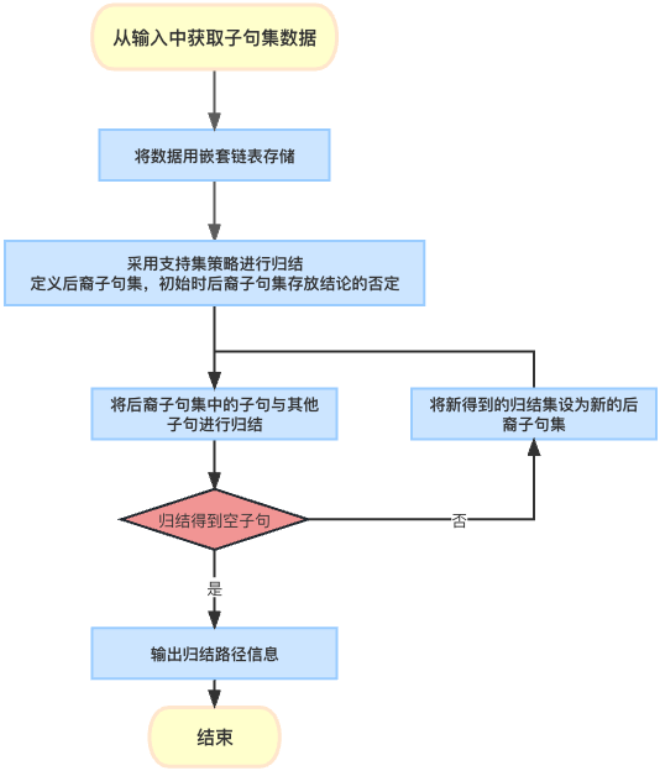
2、若 Wk 中只有一个元素，终止。并且 δk 为 W 的最一般合一，否则求出 Wk 的差异集 Dk。

3、若 Dk 中存在元素 v 和 t，并且 v 是不出现在 t 中的变量，则跳到步骤 4。否则，终止并且 W 是不可合一的。

4、更新 δk+1 和 Wk+1，即将 v 到 t 的置换添加到 δk 中并将新置换应用于 Wk。

5、置 k++并跳到步骤 2。

2. 流程图



3. 关键代码展示（带注释）

```
def find_unifier(individualities_1, individualities_2):  # 寻找合一置换
    unifiers = []
    for i in range(0, len(individualities_1)):
        if individualities_1[i] != individualities_2[i]:  # 如果个体一样则跳过
            if len(individualities_1[i]) == 1:  # 个体 1 是变量的情况
                unifiers.append([1, individualities_1[i], individualities_2[i]])  # 将置换的信息存入列表，这里还记录了是哪个子句集要置换，以防止出现两个子句集的不同变量重名的情况
            elif len(individualities_2[i]) == 1:  # 个体 2 是变量的情况
                unifiers.append([2, individualities_2[i], individualities_1[i]])
            elif (individualities_1[i][1] == '(') and (individualities_2[i][1] == '(') and (individualities_1[i][0] == individualities_2[i][0]):  # 个体 1 和个体 2 是相同函数
                if len(individualities_1[i][2:-1]) == 1:  # 个体 1 函数内是变量的情况
                    unifiers.append([1, individualities_1[i][2:-1], individualities_2[i][2:-1]])
```

```
                elif len(individualities_2[i][2:-1]) == 1:  # 个体 2 函数内是变量的情况
                    unifiers.append([2, individualities_2[i][2:-1], individualities_1[i][2:-1]])
            else:
                return None  # 不存在合一置换返回 None
    return unifiers

def displace_and_merge(unifiers, sub_resolvent_1, sub_resolvent_2):  # 进行置换和合并
    for unifier in unifiers:  # 每个置换依次处理
        if unifier[0] == 1:  # 第一个元素信息指示了是哪个子句集要置换，以防止出现两个子句集的不同变量重名的情况
            for i in range(0, len(sub_resolvent_1)):  # 两个 for 循环对所有个体进行遍历
                individualities = sub_resolvent_1[i][1]
                for j in range(0, len(individualities)):
                    individuality = individualities[j]
                    if individuality == unifier[1]:  # 如果是要置换的量即进行置换
                        sub_resolvent_1[i][1][j] = unifier[2]
        if unifier[0] == 2:
            for i in range(0, len(sub_resolvent_2)):  # 两个 for 循环对所有个体进行遍历
                individualities = sub_resolvent_2[i][1]
                for j in range(0, len(individualities)):
                    individuality = individualities[j]
                    if individuality == unifier[1]:  # 如果是要置换的量即进行置换
                        sub_resolvent_2[i][1][j] = unifier[2]
    resolvent = sub_resolvent_1 + sub_resolvent_2  # 置换后即可以进行合并
    remove_duplicates(resolvent)  # 删除重复元素
    return resolvent

def get_resolvent(clause_set_1, clause_set_2):  # 获得归结式
    for i in range(0, len(clause_set_1)):  # 两个 for 循环对所有谓词进行遍历
        predicate_1 = clause_set_1[i]
        for j in range(0, len(clause_set_2)):
            predicate_2 = clause_set_2[j]
            if '¬' + predicate_1[0] == predicate_2[0] or '¬' + predicate_2[0] == predicate_1[0]:  # 如果有互反的谓词名
                unifiers = find_unifier(predicate_1[1], predicate_2[1])  # 寻找合一置换
                if unifiers == None:  # 不存在说明无法归结
                    continue

                sub_resolvent_1 = deepcopy(clause_set_1)
                sub_resolvent_2 = deepcopy(clause_set_2)
                del sub_resolvent_1[i]  # 消去被归结掉的谓词
                del sub_resolvent_2[j]
                resolvent = displace_and_merge(unifiers, sub_resolvent_1, sub_resolvent_2)  # 置换合并后获得归结式
```

```
                return i, j, resolvent, unifiers  # 同时返回用于归结的谓词在子句中的下标
    return None, None, None, None

def resolution2(prerequisites):  # 采用支持集搜索策略进行归结
    all_clause_sets = deepcopy(prerequisites)
    path = []
    for i in range(0, len(all_clause_sets)):
        path.append([])

    start = len(all_clause_sets) - 1  # start 和 end 指针之间是目标子句的否定的后裔，初始时它们都指向目标子句的否定
    end = len(all_clause_sets) - 1
    new_end = len(all_clause_sets) - 1
    while True:
        for i in range(start, end + 1):  # 将目标子句的否定的后裔与其他子句尝试归结
            for j in range(0, i):
                which_1, which_2, resolvent, unifiers = get_resolvent(all_clause_sets[j], all_clause_sets[i])
                if resolvent != None:  # 不为 None 说明得到了归结式
                    all_clause_sets.append(resolvent)  # 把新的归结式加入到子句集的集合中
                    path.append([j, which_1, i, which_2, unifiers])  # 把归结过程的信息进行储存用于后面的输出
                    new_end += 1
                if resolvent == []:  # 归结出空子句集结束
                    return all_clause_sets, path
        start = end + 1  # 让 start 和 end 指针指向新生成的后裔
        end = new_end

def get_process(all_clause_sets, path, prerequisites):  # 得到归结路径
    process_indexs = []  # 空子句祖先的下标，即记录参与了归结推出空子句的子句
    process_indexs.append(len(path) - 1)
    add_elem_to_process(path[-1], process_indexs, path)  # 使用递归，深度优先遍历找到所有祖先
    process_indexs.sort()

    for i in range(0, len(prerequisites)):  # 把没有用到的前提条件也加入一遍进行输出
        if i not in process_indexs:
            process_indexs.append(i)
    process_indexs = list(set(process_indexs))
    process_indexs.sort()

    all_clause_sets_for_output = []  # 要输出的子句集
    path_for_output = []  # 要输出的子句集的路径信息
    for index in process_indexs:  # 为上面两个列表获取数据
        all_clause_sets_for_output.append(all_clause_sets[index])
```

```
        path_for_output.append(path[index])

    for i in range(0, len(process_indexs)):  # 开始进行输出信息的打印
        print(i + 1, ': ', end='', sep='')
        path_data = path_for_output[i]
        clause_set = all_clause_sets_for_output[i]

        if path_data == []:  # 没有路径信息说明此子句是前提条件，直接打印即可
            print_clause_set(clause_set)
        else:
            print('R[', end='')
            new_index_1 = process_indexs.index(path_data[0])  # 获得子句在输出子句集中的下标
            new_index_2 = process_indexs.index(path_data[2])
            if len(all_clause_sets_for_output[new_index_1]) == 1:  # 输出参与归结的子句的编号和哪一个文
字被归结掉
                print(new_index_1 + 1, end=',')
            else:
                print(new_index_1 + 1, end='')
                print(chr(ord('a') + path_data[1]), end=',')
            if len(all_clause_sets_for_output[new_index_2]) == 1:
                print(new_index_2 + 1, end=']')
            else:
                print(new_index_2 + 1, end='')
                print(chr(ord('a') + path_data[3]), end=']')
            if path_data[4] != []:  # 输出归结中使用的置换
                print('(', end='')
                for i in range(0, len(path_data[4])):
                    if i != 0:
                        print(',', end='')
                    unifier = path_data[4][i]
                    print(unifier[1], '=', unifier[2], sep='', end='')
                print(')', end='')
            print(' = ', sep='', end='')
            print_clause_set(clause_set)  # 打印归结式
        print('')
```

三、 实验结果及分析

1. 实验结果展示示例

测试用例 1：

```
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[6c,11b](x=w) = (¬A(w), S(w))
R[1,12a](w=tony) = S(tony)
R[2,12a](w=mike) = S(mike)
R[8b,13](z=tony) = L(tony, snow)
R[8b,14](z=mike) = L(mike, snow)
R[9a,15](u=snow) = ¬L(mike, snow)
R[16,17] = []
```

测试用例 2：

```
GradStudent(sue)
(¬GradStudent(x), Student(x))
(¬Student(x), HardWorker(x))
¬HardWorker(sue)
R[3b,4](x=sue) = ¬Student(sue)
R[2b,5](x=sue) = ¬GradStudent(sue)
R[1,6] = []
```

测试用例 3：

```
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(aa), Green(bb))
R[2,5a](x=bb,y=cc) = (¬Green(bb), Green(cc))
R[3,6a] = Green(bb)
R[4,7b] = ¬Green(bb)
R[8,9] = []
```

附加题 1：

```
I(bb)
U(aa, bb)
¬F(u)
(¬I(v), ¬U(w, v), E(w, f(w)))
(¬I(y), ¬U(x, y), F(f(z)))
R[1,5a](y=bb) = (¬U(x, bb), F(f(z)))
R[2,6a](x=aa) = F(f(z))
R[3,7](u=f(z)) = []
```

附加题 2：

```
¬P(aa)
(P(z), ¬Q(f(z), f(u)))
(Q(x, f(g(y))), R(s))
¬R(t)
R[3b,4](s=t) = Q(x, f(g(y)))
R[2b,5](x=f(z),u=g(y)) = P(z)
R[1,6](z=aa) = []
```

2. 评测指标展示及分析

两种搜索策略产生的归结式数量的比较：

	深度优先策略	支持集策略
测试用例 1	895	543
测试用例 2	15	7
测试用例 3	144	40
附加题 1	31	15
附加题 2	15	7

可见支持集策略产生的归结式数量比深度优先更少，其原因是支持集策略是一种目标制导的方向推理，其推理效率更高。在本实验中，数据没有明确哪一个子句是结论的否定，所以使用支持集策略时后裔集的初始第一条子句必须是得出空子句所必须的，而深度优先策略则不用在意这点。

四、 参考资料

课件及教材