# 中山大学计算机学院

# 人工智能

# 本科生实验报告

课程名称：Artificial Intelligence

| 学号 | 22336327 | 姓名 | 庄云皓 |
|---|---|---|---|

## 一、 实验题目

一阶逻辑的归结推理

编写函数 ResolutionFOL 实现一阶逻辑的归结推理.

该函数要点如下:

输入为子句集, KB 子句中的每个元素是一阶逻辑公式(不含等量词符号)

输出归结推理的过程, 每个归结步骤存为字符串, 将所有归结步骤按序存到一个列表中并返回, 即返回的数 据类型为 list[str] 一个归结步骤的格式为 步骤编号 R[用到的子句编号]{最一般合一} = 子句, 其中最一般合一输出格式 为"{变量=常量, 变量=常量}".如果一个字句包含多个公式, 则每个公式用编号 a,b,c 区分，如果一 个字句仅包含一个公式，则不用编号区分。

## 二、 实验内容

### 1. 算法原理

#### 1.1 基本概念

常量（constant）：任何类型的实体

如俱乐部成员：tony，mike，john　　天气类型：rain，snow

变量（variable）：如 x，y 这类未知量

项（term）：可以理解为谓词/变量的参数项，由递归定义　　变量是项（可以看成是 0 元函数）t1，t2，t3……tn 是项，f 是 n 元函数，则 f(t1,t2,,,,tn)也是项。

谓词：

零元谓词退化为命题

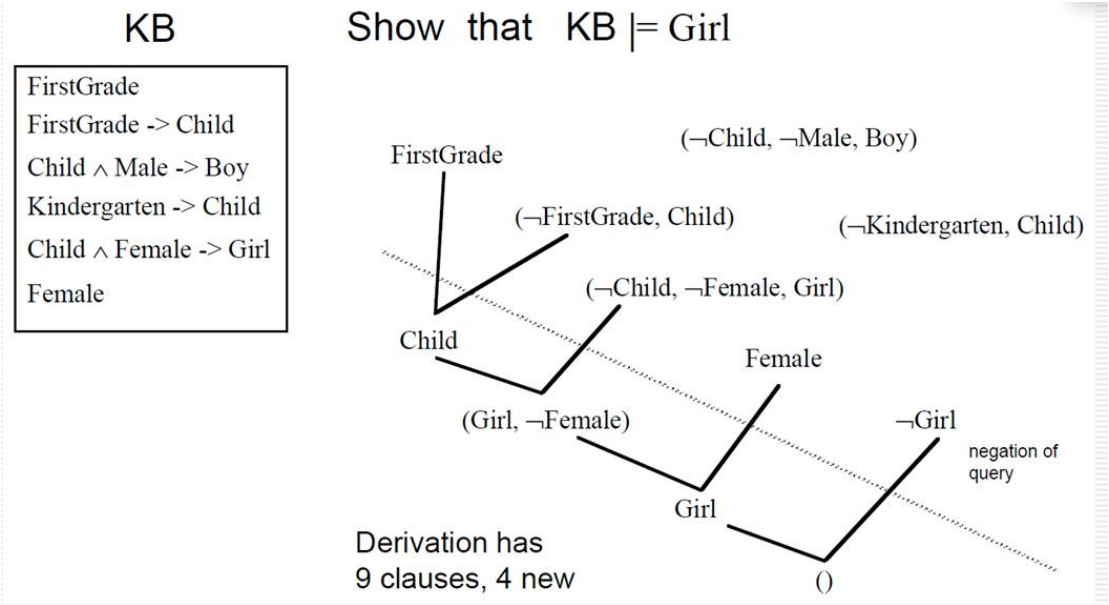
单元谓词（unary predicate）：只有一个参数，表示参数具备某 种属性，如 A(x)表示 x 属于 Alpine 俱乐部

多元谓词：有多个参数，表示参数之间的关系，如 L(x,y)表示 x 和 y 具有喜欢关系，即 x 喜欢 y

**1.2 归结算法:**KB ∧¬α 中有很多 clause 子句，我们将所有的 clause 任意两两组合，使用 resolution 进行归结。可以归结的 clause 有两个相反的 literal。

能进行归结的就进行归结得到的结果放到 new 集合中。直到 clause 无法变多，无法产生新的 clause，如果出现空子句，明 KB ∧¬α 不可满足，算法终止，可得 KB |= α，如果一直归结直到不产生新的子句，在这个过程中没有得到空子句，则 KB |= α 不成立。如下图所示

单步归结：从两个子句中分别寻找相同的原子，及其对应的原子否定去掉该原子并将两个子句合为一个，加入到 S 子句集合中例如(¬child,¬female,girl)和(child)合并为(¬female,girl)




1.3Clausal form：

是一种便于计算机处理的表达形式。这种形式下，每一条子句对应着一个元组，元组中的每一个元素是一个原子公式（或者是原子公式的否定），同时元素之间的关系是析取关系。本次实验处理的输入的子句都是这一形式。如子句¬childv¬malevboy 对应数据结构(¬child,¬male,boy)，空子句()对应 False

**1.4 合一与最一般合一**

通过变量替换使得两个子句能够被归结（有相同的原子），所以合一也被 定义为使得两个原子公式等价的一组变量替换/赋值

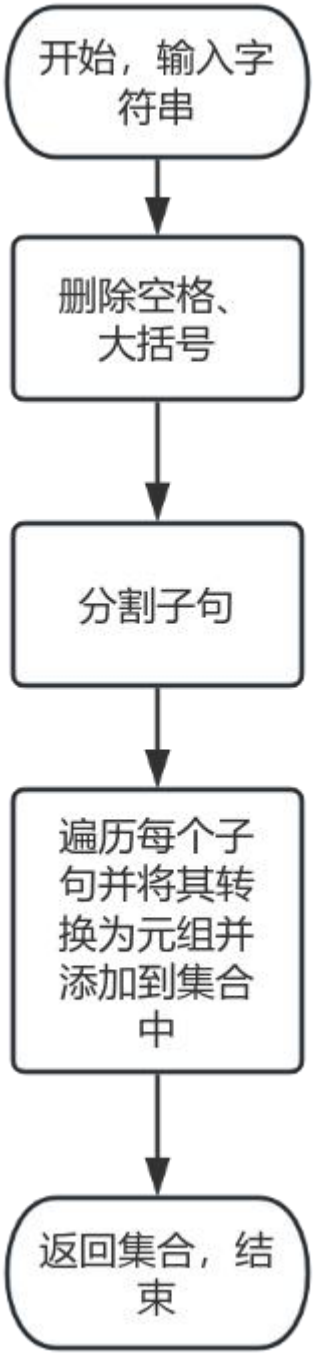
由于一阶逻辑中存在变量，所以归结之前需要进行合一，如(P(john),Q(fred),R(x))和(¬P(y),R(susan),R(y))两个子句中，我们无法找到一样的原子及其对应的否定,但是不代表它们不能够归结。通过将 y 替换为 john，我们得到了(P(john),Q(fred),R(x))和 (¬P(john),R(susan),R(john))，此时我们两个子句分别存在原子 P(john)和它的 否定¬P(john)，可以进行归结。

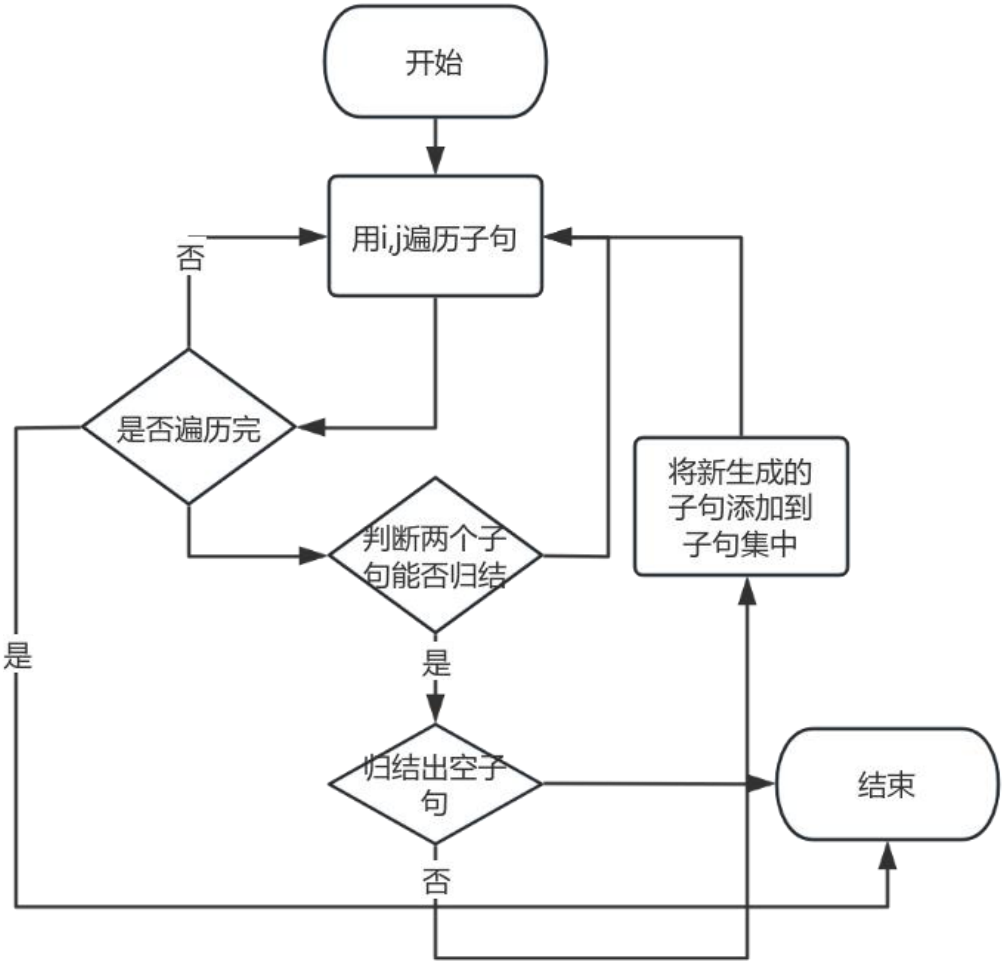
最一般合一：指使得两个原子公式等价，最简单的一组变量替换。

在本次实验中，就是先把整个字符串转化成题目要求的集合、元组的形式然后用 ij 遍历整个集合中的子句，在子句中找到前缀相同的两项，看是否能通过变量替换归结，把合并后的子句再添加到集合中。一直归结出空。然后倒过来寻找有用的子句。每个子句存储它的父节点，最后找到空子句时空子句回溯寻找有用子句

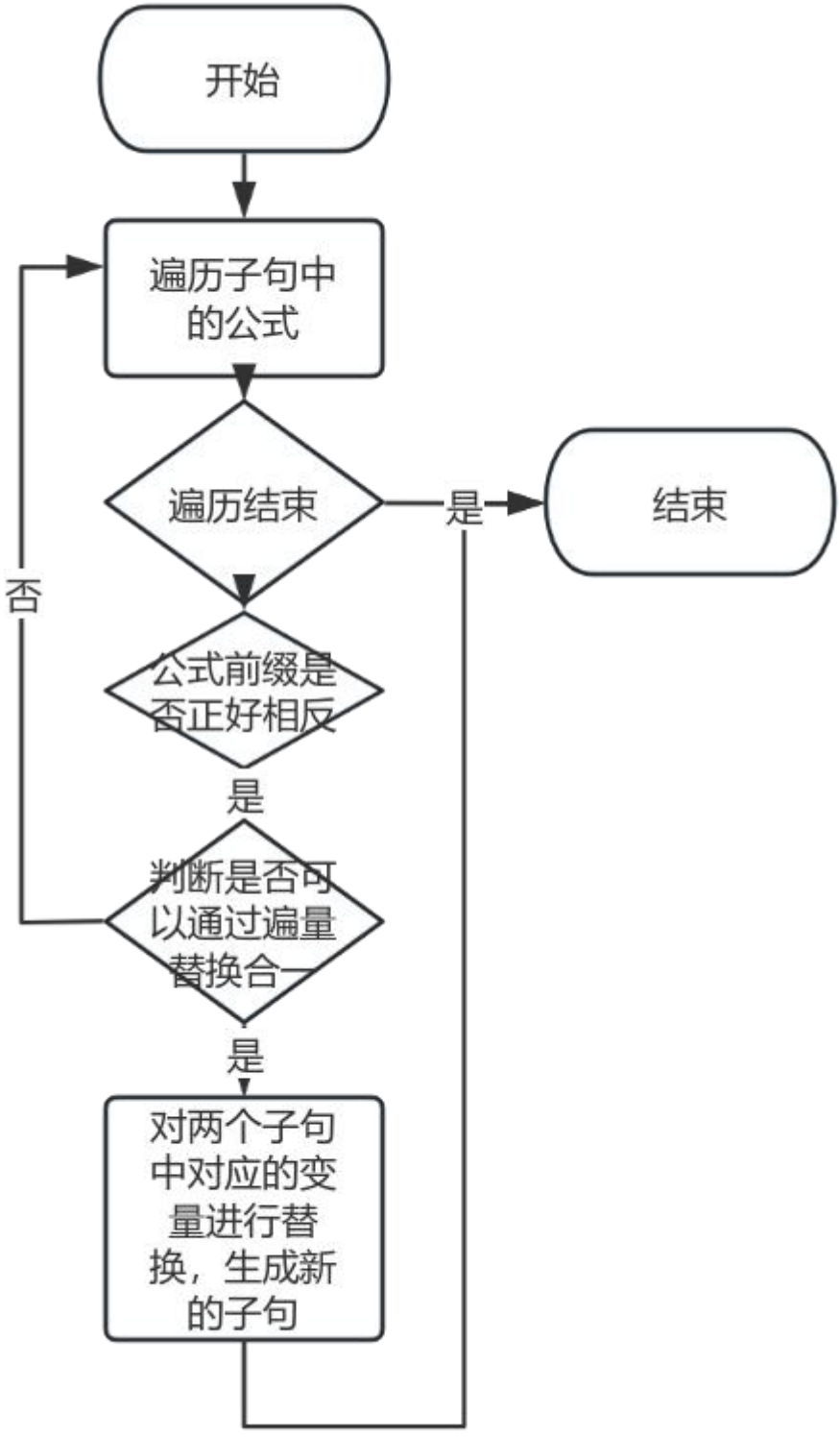
整个过程用流程图表示如下：

输入部分：






判断两个子句能否归结的流程图：




最一般合一的算法在算法原理处已经有介绍

**回溯寻找有用子句：**

因为每次生成新的子句时都用记录下他它的父节点即知道它是由哪两个子句生

成的，在回溯时对空子句进行回溯寻找它的祖先最终重新编号进行输出，具体过程见代码

## 2. 关键代码展示

两个子句的归结

```
def resolve(clause1, clause2):
    resolved_clause = []  # 存储解析后的子句
    substitution = {}  # 存储替换的变量映射
    flag = 0  # 标志变量，用于指示是否存在不一致

    # 查找第一个子句中的正文字和第二个子句中的负文字
    i, j = indexfind(clause1, clause2)

    # 如果找到正负文字
    if i != -1 and j != -1:
        literal1 = clause1[i]  # 第一个子句中的正文字
        literal2 = clause2[j]  # 第二个子句中的负文字
        # 从正文字和负文字中提取变量列表
        var_list = literal1.split("(")[1].split(")")[0].split(",")
        x_list = literal2.split("(")[1].split(")")[0].split(",")
        # 遍历变量列表，尝试进行统一化
        for var, x in zip(var_list, x_list):
            if unify(var, x, substitution) == None:  # 如果统一化失败
                flag = 1  # 设置标志为 1
                return None, None  # 返回空子句和空替换
            if substitution != None and unify(var, x, substitution) != None:
                substitution.update(unify(var, x, substitution))  # 更新替换映射
        # 创建新的子句，包含两个子句中除了正文字和负文字之外的所有文字
        for l in clause1:
            if l != literal1:
                resolved_clause.append(tihuan(l, substitution))
        for l in clause2:
            if l != literal2:
                resolved_clause.append(tihuan(l, substitution))
        # 如果生成的子句不为空，则返回子句和替换映射
        if tuple(resolved_clause) != None:
            return tuple(resolved_clause), substitution
    # 如果没有找到正负文字，返回空子句和空替换映射
    return None, None
```

```
def resolve_clauses(clauses):
    # 存储归结步骤的列表
    resolution_steps = []
```

```
count = len(clauses) + 1

# 初始化步骤编号
step_number = 1

# 复制子句以进行归结
forest_list = []
for i in clauses:
    forest_list.append(Forest(clause=i))
resolved_clauses = clauses.copy()

# 归结循环
while True:
    flag = 0
    new_clause = None

    # 遍历每对子句进行归结
    for i in range(len(resolved_clauses)):
        for j in range(i + 1, len(resolved_clauses)):
            a, b = indexfind(resolved_clauses[i], resolved_clauses[j])

            resolved = resolve(resolved_clauses[i], resolved_clauses[j])[0]
            substitution = resolve(resolved_clauses[i], resolved_clauses[j])[1]

            if substitution != {} and substitution != None:
                keys = list(substitution.keys())
                values = list(substitution.values())

            # 如果归结结果为空子句，则完成归结
            if resolved == ():
                forest_list.append(Forest(resolved, forest_list[i], forest_list[j],
                                          id1 = a if len(resolved_clauses[i])>1 else None,
                                         id2 = b if len(resolved_clauses[j])>1 else None,
                                         substitution=substitution))
                to_list(forest_list, clauses)
                ans_print(to_list(forest_list, clauses), clauses)
                resolution_steps.append(f"R[{i + 1},{j + 1}] = []")
                return resolution_steps

            # 如果归结结果不为空且未在之前的步骤中出现，则进行记录
            if resolved and resolved not in resolved_clauses and 1:  # resolved != new_clause
                new_clause = resolved
                forest_list.append(Forest(resolved, forest_list[i], forest_list[j],
```

```
                                        id1 = a if len(resolved_clauses[i])>1 else None,
                                        id2 = b if len(resolved_clauses[j])>1 else None,
                                        substitution=substitution))
                    resolved_clauses.append(new_clause)

                if i == len(resolved_clauses) - 2 and j == len(resolved_clauses) - 1 and not resolved:
                    print("不能归结出空子句")
                    return None
```

回溯寻找成有用子句：

数据结构

```
class Forest():
    def __init__(self, clause=None, pre1=None, pre2=None, id1=None, id2=None,
substitution=None):
        """
        初始化 Forest 对象。

        Args:
            clause (str): 与 forest 节点关联的子句。
            pre1 (Forest): 节点的第一个前任。
            pre2 (Forest): 节点的第二个前任。
            id1 (int): 第一个前任子句的用于归结的公式位置。
            id2 (int): 第二个前任子句用于归结的公式位置。
            substitution (dict): 与节点关联的替换字典。
        """
        self.clause = clause
        self.pre1 = pre1
        self.pre2 = pre2
        self.id1 = id1
        self.id2 = id2
        self.substitution = substitution
```

回溯过程，从最后归结出的空子句开始寻找

```
def to_list(forest, clauses):
    """
    将一个森林转换为相关节点的列表。
    Args:
        forest (list): 要转换的森林。
        clauses (list): 现有子句的列表。
    Returns:
        list: 相关森林节点的列表。
    """
    relevant = [forest[-1]]  # 从最后一个节点开始
```

```
    for node in relevant:
        if node.pre1 and node.pre1 not in relevant and node.pre1.clause not in
clauses:
            relevant.append(node.pre1)
        if node.pre2 and node.pre2 not in relevant and node.pre2.clause not in
clauses:
            relevant.append(node.pre2)
    relevant.reverse()  # 反转列表以保持顺序
    return relevant
```

**3. 创新点&优化**

在寻找可以合一的子句时可以从较短的句子入手，能更快归结出空子句，例如可以用一个优先队列存储子句，句子的长度为优先级。由于这时我的代码已经快写好且这个改动较大，遂放弃。

## 三、 实验结果及分析

**1.** 实验结果展示示例

```
==============TEST0===============

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

==============TEST1===============

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

```
==============TEST2===============

1 ('On(mike,john)',)
2 ('Green(tony)',)
3 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
4 ('~Green(john)',)
5 ('On(tony,mike)',)
6 R[1,3a]{xx=mike,yy=john} = ('~Green(mike)', 'Green(john)')
7 R[3c,6a]{yy=mike} = ('~On(xx,mike)', '~Green(xx)', 'Green(john)')
8 R[4,7c] = ('~On(xx,mike)', '~Green(xx)')
9 R[5,8a]{xx=tony} = ('~Green(tony)',)
10 R[2,9] = ()
PS D:\Data\作业\大二下作业\ai2024>
```

**2. 评测指标展示及分析**

本次实验的程序主要的时间复杂度集中在遍历子句进行归结以及合一这一阶段上，回溯寻找有用子句是的时间复杂度小于这一阶段的，整体程序的时间复杂

**度可以近似认为就是这一阶段的时间复杂度。**

设子句集有 m 个子句，每个子句有 n 个公式，则当每个子句的原子公式数量比较少（只有个位数）的情况下（即本次实验给出的三个例子），程序的时间复杂度可以近似认为为$O(m^2n^2)$,当每个子句的原子公式数量比较多时，设合一时遍历的时间复杂度为$O(n)$,每个谓词的平均参数数量为 q 总时间复杂度为$O(m^2n^3q)$,实际由于还有变量替换等原因时间复杂度应该更大。[1]

然而外层循环没有只遍历一次，一阶逻辑的归结推理是 NP 完全问题，时间复杂度最坏情况下可以达到指数级别。

# 四、 参考资料

[1] 【人工智能导论】实验二 一阶逻辑归结算法 - 哔哩哔哩 (bilibili.com)
[2] 3-归结原理.pdf
[3] Ch2-知识表示与推理-II.pdf