中山大学计算机学院
人工智能
本科生实验报告
（2024 学年春季学期）

课程名称：Artificial Intelligence

教学班级		专业（方向）	
学号		姓名	

一、 实验题目

命题逻辑和一阶逻辑的归结算法以及最一般合一算法

二、 实验内容

1. 算法原理

一、命题逻辑的归结算法原理

Step1：将要证明的命题取否定后加入到 KB 中，再将 KB 转换成 clausal form 的形式得到子句集 S.

- 子句集是由一个或者多个子句的合取构成的集合
- 子句是由一个或者多个文子的析取构成

Step2：使子句集反复的进行单步归结，直到产生空子句（NIL）或者没有新的子句产生

- 单步归结：
  - 从一个子句C_1中找到一个原子公式 L，同时在另一个子句C_2中找到与原子公式 L 互补的文字$\neg L$
  - 分别删去两个子句C_1和C_2中的 L 和¬L，并将两个子句剩下的原子公式合并成一个新的子句加入到子句集 S 当中

二、最一般合一（MGU）算法原理

Step1：初始化$k=0$，$\sigma_k=\{\}$，$S_k=\{f，g\}$，$D_k=\{\}$

Step2：如果当前 f 和 g 是相同的原子公式，即谓词和所有的项都相同，那么算法终止，此时的置换σ_k就是 f 和 g 最一般合一的结果

Step3：否则找出两个公式的差异集$D_k=\{e_1，e_2\}$

- 如果$e_1=v$是一个变量且$e_2=t$是一个不包含v的项，那么就更新置换集和子集：$\sigma_{k+1}=\sigma_k\circ\{t/v\}$，$S_{k+1}=S_k\sigma_{k+1}$，$(t/v$即$v=t)$重复步骤 Step2
- 否则，算法终止，f，g不存在最一般合一

三、一阶逻辑的归结算法原理

Step1：将要证明的命题取否定后加入到 KB 中，再将 KB 转换成 clausal form 的形式得到子句集 S.

Step2：使子句集反复的进行单步归结，直到产生空子句（NIL）或者没有新的子句产生

- 单步归结：
  - 从一个子句C_1中找到一个谓词为 P 的原子公式 L，同时在另一个子句C_2中找到与原子公式 L 谓词相同的互补的文字$\neg L$

- 使用 MGU 算法对谓词相同且互补的原子公式获得合一的置换集σ，同时使得两个子句分别应用置换集σ，即$C_1^{'} = C_1\sigma$，$C_2^{'} = C_2\sigma$
- 分别删去两个子句$C_1^{'}$和$C_2^{'}$中的 L 和¬L，并将两个子句剩下的原子公式合并成一个新的子句加入到子句集 S 当中

2. 伪代码

一、命题逻辑的归结算法伪代码

Algorithm 1: Propositional Logic Resolution

```
function ResolutionProp(KB)
    inputs: KB      //子句集
    returns: steps  //归结步骤列表
1.   var clause_dict:dict
     var clauses,steps:list
     var generations//记录归结代数
     var parent_info:dict  //记录子句亲本信息
2.   while true do
3.       for pre_generation in generations do
4.           for clause1 in current_generation do
5.               for clause2 in pre_generation do
                     //当前代一次与之前的所有代一一配对归结
6.                   formula1,formula2 ← find_formula_pair
7.                   if formula1,formula2 then  //如果找到互补对
8.                       new_clause ← resolve(new_clause1,new_clause2)//归结
9.                       if new_clause not in clausese then  //如果是新子句
                             //更新新子句和归结步骤
10.                          update(clauses,steps,parent_info)
11.                          if new_clause == () then   //如果是空子句
                                 //生成归结路径
12.                              resolution_path ← generate_resolution_path
13.                              steps ← generate_steps  //生成归结步骤
14.                              return steps           //找到矛盾
15.      if not next_generation then
             break
16.      generations ← next_generation
17.  return steps
//寻找两个子句中原子公式互补对
function find_formula_pair(clause1,clause2)
    inputs: clause1,clause2         //两个子句
    returns: (formula1,formula2)    //互补对或(null, null)
1.   for (formula1,formula2) in (clause1,clause2) do
2.       if complementary_pair(formula1,formula2) then //如果 f1 和 f2 互补
           return (formula1,formula2)
3.   return (null,null)
```

```
        //归结操作
function resolve(clause1,clause2,formula1,formula2)
    inputs: clause1,clause2     //两个子句
            formula1,formula2   //要消去的互补对
    returns: new_clause         //归结后的新子句
1.  return (clause1 ∪ clause2)-{formula1,formula2} //集合的差集操作
```

二、最一般合一（MGU）算法伪代码

Algorithm 2: MGU (Most General Unify)

```
function MGU(formula1,formula2)
    inputs: formula1, formula2  // 要合一的原子公式
    returns: σ                  // 最一般合一替换集
1.   D ← []                     // 初始化差异集
2.   σ ← {}                     // 初始化替换集
3.   while true do
4.       D ← difference(formula1, formula2) //difference 函数得到差异集
         //D 中的元素有可能为原子公式，个体变量，个体常量
5.       if D == [] then    //不存在差异集退出
            break
6.       for items in D do  //对于 D 中的元素,令 s 为变量类型，t 可以是任何形式
7.           if items is variable then
8.               s=items,t=another
9.       if (s is variable) ∧ (s is not in t) then  //s 是变量且不在 t 中出现
10.          σ ← σ ◦ {s:t}  //使得置换{s:t}与σ进行复合
11.      else then          //其他情况下直接退出
             break
12.      formula1 ← formula1·σ  //应用置换集
13.      formula2 ← formula2·σ
14.  return σ
function difference(formula1, formula2)
    inputs: formula1, formula2  //要比较的两个原子公式
    returns: D                  //差异集
1.   pred1, terms1 ← extrct(formula1)
2.   pred2, terms2 ← extrct(formula2)
3.   if pred1 ≠ pred2 then
         return [formula1, formula2]
4.   if len(terms1) ≠ len(terms2) then
5.      return [formula1, formula2]
6.   for i ← 1 to len(terms1) do
7.       if difference(terms1[i],terms2[i]) then
8.           differences.append(difference(terms1[i],terms2[i]))
9.   return differences[0]
function extrct(formula)
```

```
    inputs: formula //需要提取谓词和项的原子公式
    returns: predicate,terms //谓词和项
1.   left ← formula.find('(')
2.   right ← formula.rfind(')')
     //单个变量无谓词返回空谓词以及自己本身就是自己唯一的项
3.   if left == -1 or right == -1 then
        return '',formula
     //若有否定词,则从第二个字符到左括号的位置为谓词
4.   if formula.startswith('~') then
5.      predicate ← formula[1:left]
     //其他情况从第一个字符到左括号为谓词
6.   else then
7.       predicate ← formula[0:left]
     //将项字符串按照逗号分割转换成列表
8.   terms ← [term for term in formula[left+1:right].split(',')]
9.   return predicate,terms
```

三、一阶逻辑的归结算法伪代码

Agorithm3: First Order Logic Resolution

```
function FirstOrderLogicResolution(KB)
    inputs:KB       //一阶逻辑子句集合
    returns: steps  //归结步骤列表
1.   var clause_dict:dict
     var clauses,steps:list
     var generations//记录归结代数
     var parent_info:dict  //记录子句亲本信息
2.   while true do
3.       for pre_generation in generations do
4.           for clause1 in current_generation do
5.               for clause2 in pre_generation do
                     //当前代一次与之前的所有代一一配对归结
                     //获得两个子句的所有可能的合一置换集
6.                   substitution_ways ← MGU_clause(clause1,clause2)
                     //应用置换集
7.                   for σ in substitution_ways do
8.                       new_clause1,new_clause2 ← clause1·σ,clause2·σ
                         //寻找互补对
9.                       formula1,formula2 ← find_formula_pair
10.                      if formula1,formula2 then  //如果找到互补对
                         //归结
11.                      new_clause ← resolve(new_clause1,new_clause2)
12.                      if new_clause not in clauses then //如果是新子句
                             //更新新子句和归结步骤记录亲本信息
```

```
11.                          update(clauses,steps,parent_info)
12.                          if new_clause == () then   //如果是空子句
                                 //生成归结路径
13.                              resolution_path ← generate_resolution_path
14.                              steps ← generate_steps  //生成归结步骤
15.                              return steps           //找到矛盾
16.      if not next_generation then
             break
17.      generations ← next_generation
18.  return steps
//生成归结路径
function generate_resolution_path(new_clause, parent_info) → resolution_path
    inputs: new_clause      //空子句
            parent_info     //亲本信息
    returns: resolution_path //归结路径
//生成步骤信息
function generate_steps(resolution_path, clause_dict, parent_info, KB) → steps
    inputs: resolution_path  //归结路径
            clause_dict      //子句编号字典
            parent_info      //亲本信息
            KB              //初始子句集
    returns: steps          //步骤列表
//寻找两个子句中原子公式互补对
function find_formula_pair(clause1,clause2) → (formula1,formula2)
    inputs: clause1,clause2         //两个子句
    returns: (formula1,formula2)    //互补对或(null, null)
//归结操作
function resolve(clause1,clause2,formula1,formula2) → new_clause
    inputs: clause1,clause2     //两个子句
          formula1,formula2    //要消去的互补对
    returns: new_clause         //归结后的新子句
//计算子句的所有可能的置换集
function MGU_clause(clause1, clause2) → list of substitutions
    inputs: clause1, clause2         //两个子句
    returns: list of substitutions   //所有可能的 MGU 置换
//应用置换集
function replace_clause(clause,σ) → replaced_clause
    inputs: clause      //要处理的子句
            σ          //要应用的替换集
    returns: new_clause
```

3. 关键代码展示

本部分将介绍最一般合一算法以及一阶逻辑归结算法的代码实现，由于一阶逻辑归结算法的实现代码是在命题逻辑算法代码上改进补充得到的，一阶逻辑归结的代码完全可以实现命题逻辑的归结，故为了避免重复赘述，这里将展示和介绍最一般合一算法以及一阶逻辑归结算法的代码实现。

一、最一般合一算法代码与设计思路

代码框架与核心函数实现：首先我们根据第 2 部分中的最一般合一算法原理，可以得到 MGU 核心实现函数的基本代码框架：

①首先需要有两个子句的输入 `formula1` 和 `formula2`，同时需要有设置差异集 `D` 和置换集 `substitution`

②使用 `difference` 函数获得两个原子公式的差异集 `D`，

- 如果 `D` 为空那么退出循环
- 如果 `D` 不为空那么，将 `D` 中属于变量的部分赋值给 `v`，另一个赋值为 `t`，如果不存在变量那么两个公式不存在合一退出循环

③使用 `extract` 函数提取出原子公式 `t` 的项列表 `terms`，

- 如果 `v` 不在 `t` 的项中出现那么，使用 `composition` 函数对 `substitution` 与`{v:t}`进行置换集的复合运算，同时使用函数 `replace_variable` 对当前的原子公式 `formula1` 和 `formula2` 使用`{v:t}`置换

具体的实现代码如下：

```python
def MGU(formula1,formula2):
    D=[]            #差异集D
    substitution={} #替换集σ
    while 1:
        #使用difference函数获取两个原子公式的差异集
        D=difference(formula1,formula2)
        #如果差异集为空，那么循环结束
        if D==[]:
            break
        #遍历差异集中的两个元素e1和e2
        '''
            使用正则表达式来判断一个字符串类型的项是不是变量
            这里我们暂时认为只有字母d-z的单个形式或者重复的两个为变量
            如 x,y,z,xx,yy,zz等
        '''
        #如果D[0]为变量
        if re.fullmatch(r'^([d-z])(\1)?$', D[0]):
            v,t=D[0],D[1]
        #如果D[1]为变量
        elif re.fullmatch(r'^([d-z])(\1)?$', D[1]):
            v,t=D[1],D[0]
        #如果两个都不是变量那么表示不存在合一直接退出
        else:
            break
        #提取t的谓词和项，这里为了方便认为变量x的谓词为''项只有一个那就是他本身
        predicate,terms=extrct(t)
        #如果v不在t的项中
        if v not in terms:
            #使σ与{v:t}复合
            substitution=composition(substitution,{v:t})
            #条件子句都应用新的替换集
            formula1=formula1.replace(v,t)
            formula2=formula2.replace(v,t)
    return substitution
```

重要函数模块的实现：为了实现核心函数所设立的框架，我们需要依次对框架里里面所提到函数进行一一实现

①replace_variable 和 composition 函数：**应用置换集**的函数和**置换集的复合**函数代码实现和注释如下：利用正则表达式的匹配以及使用字典的运算实现

```
def replace_variable(formula,substitution):
    new_formula=formula
    #遍历置换集字典的每一个items
    for old,new in substitution.items():
        #使用正则表达式将原子公式中所有的old都替换成new
        #正则表达式的好处就是不会错误的替换原子公式中的字串
        #比如要将w替换成a，那么他不会将jown中的w替换成a
        pattern = r'\b' + re.escape(old) + r'\b'
        new_formula=re.sub(pattern,new,new_formula)
    return new_formula

def composition(set1,set2):
    set={}
    #两两作用的到原始的替换集
    for old,new in set1.items():
        set.update({old:replace_variable(new,set2)})
    #删除掉有歧义的替换，即替换是一一对应的
    for old,new in set2.items():
        if old not in set:
            set.update({old:new})
    #删掉old=new的替换
    return {old:new for old,new in set.items() if old!=new}
```

②difference 函数：**计算两个原子公式的差异集 D** 的函数实现如下：值得注意的是 difference 函数采用了递归的办法实现，这样就可以获得两个原子公式真正的差异的地方，比如 P(g(x))和 P(g(h(x)))的差异是 x 和 h(x)并不是 g(x)和 g(h(x))

```
def difference(formula1,formula2):
    #提取原子公式的terms
    predicate1,terms1=extract(formula1)
    predicate2,terms2=extract(formula2)
    #如果两个公式都是变量或者常量
    if is_individual(formula1) and is_individual(formula2):
        #如果两个原子公式相等，那么递归结束返回
        if formula1==formula2 :
            return
        else:
        #否则就说明这就是两个公式差异的地方，返回差异集
            return [formula1,formula2]
    #如果谓词不相等，那么就是差异的地方
    if predicate1!=predicate2:
        return [formula1,formula2]
    elif predicate1 == predicate2:
        differences=[]
        #如果谓词相等，那么递归的调用函数比较两个原子公式的项之间的差异
        for i in range(len(terms1)):
            if difference(terms1[i],terms2[i]):
                differences.append(difference(terms1[i],terms2[i]))
    if len(differences)>0:
        return differences[0]
    else:
        return []
```

③extract 函数：**提取原子公式的谓词和项列表**的函数的实现如下，这个函数的功能是提取出一个原子公式的谓词和项列表，比如~P(x,y)的谓词是 P，项列表是[x,y]，这里为了方便统一处理我们不严谨的认为单个变量的谓词为空字符串，而唯一的项就是他自己本省，比如 x 的谓词为'',项列表为[x,]

```
def extract(formula):
    '''
    从原子公式中提取谓词和参数项
    如果公式是单个变量，则返回空字符串和该变量
    args(str):
    formula:原子公式字符串
    return(str,list):
    predicate,terms:谓词,参数项列表
    '''
    left=formula.find('(') #from left to right find
    right=formula.rfind(')') #from right to left find
    if left==-1 or right==-1:
        return '',formula #单个变量无谓词返回空谓词以及自己本身就是自己唯一的项
    #如果原子公式以~开头，那么谓词就是从第二个字母开始到左括号的子串
    if formula.startswith('~'):
        predicate=formula[1:left]
    #如果是正文字，那么谓词就是从第一个字母到左括号的字串
    else:
        predicate=formula[0:left]
    #原子公式的项就是左括号到右括号的字串同时用逗号分割成的列表
    terms=[term for term in formula[left+1:right].split(',')]
    return predicate,terms
```

二、一阶逻辑的归结算法代码与设计思路

代码框架与核心函数的实现：同样的根据一阶逻辑归结算法原理，我们可以得出核心函数的实现框架：

①将上一轮归结的代数 current_generation 取出，同时遍历过往的每一代 pre_generation，clause1 从当前代中遍历，clause2 从过往代 pre_generation 中遍历，如果两个子句相同我们进行减支处理直接跳过

②使用 MGU_clause 函数获得两个子句所有可能的置换方式，因为两个子句的不同原子公式之间存在不同的合一，我们将不同原子公式的合一的置换集记录下来，一一尝试

③遍历两个子句的所有置换方式，对两个亲本子句进行置换集置换，同时使用 find_formula_pair 函数来寻找两个子句之间的互补原子公式

- 如果没有找到了互补的原子公式对，那么直接跳过后面的过程
- 如果找到了互补的原子公式，我们使用 resolve 对两个置换过后的子句进行归结操作获得新的子句 new_clause

④判断 new_clause 是否出现过

- 如果 new_clause 出现过，那么也可以跳过后面过程
- new_clause 没有出现过，那么将他加入到总的子句集，同时记录他的亲本信息，包括亲本子句，使用的置换集，消去的原子公式对，这些信息将会用于后续回溯出归结路径
- 如果 new_clause 是空子句()，那么就表明推出了矛盾，算法结束，使用 generate_resolution_path 生成从条件子句到空子句的归结路径，使用 generate_steps 函数产生规整的归结步骤

具体的代码实现框架如下：

```
def FirstOrderLogicResolution(KB):
    while 1:
        current_generation=generations[-1]
        next_generation=[]
        #取出子句集中的两个不重复的子句
        new_clause=()
        for pre_generation in generations:     #遍历过往的每一代
            for clause1 in current_generation: #clause1从当代中寻找
                for clause2 in pre_generation: #clause2从过往的代数寻找
                    if clause1 == clause2:
                        continue
                    #获得两个子句的所有可能的置换的方法，一次归结只能够合一两个子句的两个原子公式
                    substitution_ways=MGU_clause(clause1,clause2)
                    #遍历所有的可能的两个子句的置换
                    for sub in substitution_ways:
                        #应用置换集获得置换后的子句
                        new_clause1=replace_clause(clause1,sub)
                        new_clause2=replace_clause(clause2,sub)
                        #寻找两个子句是否存在互补的原子公式
                        formula1,formula2=find_formula_pair(new_clause1,new_clause2)
                        #没有原子公式那么跳过剩下的步骤
                        if not formula1:
                            continue
                        #对置换后的子句进行归结
                        new_clause=resolve(new_clause1,new_clause2,formula1,formula2)
                        #如果归结出了新的子句
                        if new_clause not in clauses:
                            clauses.append(new_clause)      #加入到子句集最后 广度优先
                            #clauses.insert(0,new_clause)   #加入到子句集最前 深度优先
                            parent_info[new_clause]={
                                'parents':(clause1,clause2),
                                'sub':sub,
                                'formulas':(formula1,formula2)
                            }#记录该子句的亲本信息，包括亲本子句，置换以及互补的原子公式
```

```
        #如果新的子句是空子句说明归结完成
        if new_clause == ():
            #生成归结路径，过滤掉无用子句
            resolution_path=generate_resolution_path(new_clause,parent_info)
            clause_dict={clause:index+1 for index,clause in enumerate(KB)} #编号
            #生成归结步骤用于输出
            steps=generate_steps(resolution_path,clause_dict,parent_info,KB)
            return steps
```

重要函数模块的实现：为了实现核心函数所设立的框架，我们需要依次对框架里所提到函数进行一一设计和实现

①MGU_clause 和 replace_clause 函数：MGU 的合一置换都是针对原子的，这里是针对子句的合一置换。MGU_clause **计算子句的合一方式**，考虑子句中不同原子公式的合一的不同结果；replace_clause 函数实现了**对子句中所有的原子公式进行置换**。

```
def MGU_clause(clause1,clause2):
    substitution_ways=[{}]
    #遍历两个子句中所有的原子公式组合
    for formula1 in clause1:
        for formula2 in clause2:
            #调用MGU函数实现两个原子公式的最一般合一置换集
            #加入到置换方法列表中
            substitution_ways.append(MGU.MGU(formula1,formula2))
    return substitution_ways

def replace_clause(clause,substitution):
    if not substitution:
        return clause
    new_clause=[]
    clause=list(clause)
    #遍历子句的每个原子公式
    for formula in clause:
        #调用MGU中的replace_variable函数，将每个原子公式都进行置换
        new_clause.append(MGU.replace_variable(formula,substitution))
    return tuple(new_clause)
```

②find_formula_pair 函数：**得到两个子句中的存在的互补原子公式对**的函数实现如下，遍历一个子句的所有的原子公式，如果是以~的负文字那么就找另一个子句中是否存在对应的正文字，如果是正文字那么就寻找对应的负文字。

```
def find_formula_pair(clause1,clause2):
    '''
    从两个子句中分别寻找原子公式互补对：
    包括相同的原子公式及其对应的原子公式否定
    args(tuple,tuple):
        clause1: 子句1
        clause2: 子句2
    return(str,str):
        找到的原子公式互补对
        如果不存在则返回None
    '''
    for clause_formula in clause1:
        if ('~'+clause_formula) in clause2:
            return clause_formula,'~'+clause_formula
        elif clause_formula.startswith('~') and clause_formula[1:] in clause2:
            return clause_formula,clause_formula[1:]
    return None,None
```

③resolve 函数：用于**实现两个子句的归结**的函数实现如下，利用 python 中 set 中的并集和差集运算，将两个子句的所有原子公式并起来，之后再去掉中间互补的原子公式。

```
def resolve(clause1,clause2,formula,complt_formula):
    '''
    实现单步归结操作
    args(tuple,tuple,str,str):
        clause1: 子句1
        clause2: 子句2
        formula,complt_formula: 要消去的原子公式及其否定
    returns(tuple):
        new_clause:归结新产生的子句
    '''
    new_clause=[]
    #利用set实现差集的运算
    new_clause=tuple(sorted((set(clause1+clause2))-{formula,complt_formula}))
    return new_clause
```

④generate_resolution_path 函数：生成归结路径的函数实现如下，由于我们存储了子句的亲本子句，那么这时的归结路径可以类比于一个倒立的二叉树，所以采用二叉树中广度优先的方法从空子句向上不断回溯亲本子句直到回溯到初始条件子句，这时将 resolution_path 翻转再保留顺序去重，得到从条件到空子句的归结路径，这时的归结路径是没有多余的无用子句的。

```
def generate_resolution_path(new_clause,parent_info):
    resolution_path=[]
    #采用BFS的方法从空子句()回溯到初始条件获得归结路径，这时归结路径是反着的
    queue=deque()
    queue.append(new_clause)
    while len(queue)>0:
        current_clause=queue.popleft()
        resolution_path.append(current_clause)
        for parent_clause in parent_info[current_clause]['parents']:
            if parent_clause is not None:
                queue.append(parent_clause)
    #将归结路径reversed后，转成字典后再转成list的方法去重，这种方法可以保证去重前后的顺序不会改变
    resolution_path=list(dict.fromkeys(reversed(resolution_path)))
    return resolution_path
```

⑤generate_steps 函数：**将归结路径转换成规整的归结步骤输出**的函数实现如下，遍历归结路径的所有子句，将之前记录的亲本信息取出来，对亲本子句进行替换，然后我们为亲本子句的所有原子公式进行编号，如果有多个原子公式，那么用 a,b,c 来加以区分。然后使用 f'格式生成规整的归结步骤放入到 steps 中。

```
def generate_steps(resolution_path,clause_dict,parent_info,KB):
    steps=[]
    #将子句条件加入步骤列表
    for i,clause in enumerate(KB):
        steps.append(clause)
    #从头遍历归结路径
    for clause in resolution_path:
        if clause not in list(KB):
            #获得亲本子句的信息，包括亲本子句，置换集以及消去的互补原子公式
            clause1,clause2=parent_info[clause]['parents']
            sub=parent_info[clause]['sub']
            formula1,formula2=parent_info[clause]['formulas']
            #应用置换集获得置换后的子句
            new_clause1=replace_clause(clause1,sub)
            new_clause2=replace_clause(clause2,sub)
            #为子句的每一个项进行编号，用于后面的规格化输出
            clause1_dict=numbering_formula(new_clause1,clause_dict[tuple(clause1)])
            clause2_dict=numbering_formula(new_clause2,clause_dict[tuple(clause2)])
            #利用差集的进行归结
            new_clause=resolve(new_clause1,new_clause2,formula1,formula2)
            #为新子句进行编号
            clause_dict.update({new_clause:len(clause_dict)+1})
            #规格化步骤的格式即 步骤号R[亲本子句的编号]{应用的置换} = 归结后的子句
            if not sub:
                new_step=f'R[{",".join(sorted(map(str,[clause1_dict[formula1],clause2_d
            else:
                substitution_str='{'+','.join([f'{old}={new}' for old, new in sub.items
                new_step=f'R[{",".join(sorted(map(str,[clause1_dict[formula1],clause2_d
            steps.append(new_step)
    return steps
```

4. 创新点&优化

在理论课的学习当中有专门的一小节介绍了一些归结策略，（具体可以去查看第三讲 PPT 第 78 页），其中里面用的最多的包括本次实验所用的归结策略使用的是宽度优先的策略，宽度优先的归结策略是完备的同时也是能够产生最优归结的一种策略，因为在这种策略下空子句在归结演绎树下的深度是最小的，如果归结成功的话，那么缺点就是容易产生组合爆炸，导致我们产生了很多无用的归结子句。

在以前程序设计的学习中可以发现有一个规律，有 bfs 算法和 dfs 算法基本上是成对出现的，所以这里我思考着是否能够使用深度优先的归结策略进行归结。

深度优先的归结策略可以参考宽度优先，宽度优先的其实就是当我们产生了新的子句之后，我们将新子句放在总的子句集的最后，在前面的子句都考虑完之后再考虑新的子句与其他子句的组合，那么深度优先的策略就可以把新归结出来的子句放到子句集的最前面，每次归结都优先考虑上一次新归结出来的子句是否与其他的子句有新的组合：

new_clause：(1,2) → 4　　(2,3) → 5

宽度优先策略：[1,2,3,4,5]

深度优先策略：[5,4,1,2,3]

深度优先的归结策略优先使用了最先归结出来的子句，在实际的归结中这点有利于提高归结的效率的，同时他也减少了归结子句的产生，但是缺点就是他所产生的归结路径不一定是最优的，实际上如果只考虑是否归结成功，dfs 是一种高效的选择，在第三部分实验结果分析中给出了四种不同的归结策略的产生的归结子句和归结深度的比较。

三、 实验结果及分析

1. 实验结果展示

一、命题逻辑归结运行结果

当输入 KB = {("FirstGrade",),("~FirstGrade","Child"),("~Child",)} 可以获得如下的输出：可以看到我的输出符合实验要求的输出。

```
Input:
KB =  {('~FirstGrade', 'Child'), ('FirstGrade',), ('~Child',)}
Output:
1 ('~FirstGrade', 'Child')
2 ('FirstGrade',)
3 ('~Child',)
4 R[1a,2] = ('Child',)
5 R[3,4] = ()
PS C:\Users\刘天翔\Desktop\python>
```

二、MGU 代码运行结果

输入('P(xx,a)','P(b,yy)')和'P(a,xx,f(g(yy)))','P(zz,f(zz),f(uu))'可以获得如下的输出：可以看到我们的输出符合实验要求，同时我们也满足着只有变量能够被替换的理论要求，也就是说只有变量才能够作为字典的键。

```
PS C:\Users\刘天翔\Desktop\python> & D:/Anaconda/envs/pytorch2.
验2/Code/MGU.py
Input: P(xx,a) P(b,yy)
Output:
{'xx': 'b', 'yy': 'a'}

Input: P(a,xx,f(g(yy))) P(zz,f(zz),f(uu))
Output:
{'zz': 'a', 'xx': 'f(a)', 'uu': 'g(yy)'}
PS C:\Users\刘天翔\Desktop\python>
```

三、一阶逻辑归结运行结果

查看我们算法的输出，可以看到我们使用了从空子句到初始条件子句的方式可以让我们得到最短最优的归结路径用于我们的输出，这种算法的好处就是他不会产生除了归结路径以外无关的归结步骤的输出，符合着实验任务中所要求的输入。由于实验规定我们用于输入的子句集 KB 一定是用 set 的数据结构存储，由于 set 的无序性我们无法确定我们输入到函数中的子句集的顺序是什么样子的，但是这种无顺序的影响我们程序输出最优的归结过程。

输入下面三个子句集：

```
KB1={('GradStudent(sue)',),('~GradStudent(x)','Student(x)'),
    ('~Student(x)','HardWorker(x)'),('~HardWorker(sue)',)}
KB2={('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)')}
KB3={('On(tony,mike)',),('On(mike,john)',),('Green(tony)',),
    ('~Green(john)',),('~On(xx,yy)','~Green(xx)','Green(yy)')}
```

可以获得如下的三组输出：

例题：

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

作业一：

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

作业二：

```
Output:
1 ('Green(tony)',)
2 ('~Green(john)',)
3 ('On(tony,mike)',)
4 ('On(mike,john)',)
5 ('~On(xx,yy)', '~Green(xx)', 'Green(yy)')
6 R[2,5c]{yy=john} = ('~Green(xx)', '~On(xx,john)')
7 R[1,5b]{xx=tony} = ('Green(yy)', '~On(tony,yy)')
8 R[4,6b]{xx=mike} = ('~Green(mike)',)
9 R[3,7b]{yy=mike} = ('Green(mike)',)
10 R[8,9] = ()
----------------
```

2. 评测指标展示及分析

下表比较了四种不同的归结策略所产生的子句集数量，以及归结的深度，同时使用了相同的子句集进行测试，前三种策略具体可以查看第三讲 PPT 的 P78

简单子句集采用 PPT 上的例子：

```
('~I(x)','R(x)'),('I(a)',),('~R(y)','~L(y)'),('L(a)',)
```

复杂子句集采用作业二的子句集

相关数据如下表所示，

归结策略	简单子句集		复杂子句集	
	产生的子句数量/个	获得空子句的深度/层	产生的子句数量/个	获得空子句的深度/层
广度优先	9	2	149	3
线性输入	9	3	220	6
支持集	7	3	152	6
深度优先	7	2	24	12

可以看到**深度优先的策略可以有效的降低归结过程中产生的子句的数量**，数量减少也就减少了归结所产生的时间，提高了归结的效率，这一点总体上随着初始子句集条件的数量的增加，体现的越明显。

同时也可以看到**深度优先的策略可能会导致归结的深度增加**，所有会导致我们所得出的归结路径和步骤不是最优的，也就是可能会绕一下圈子，这一现象随着初始条件子句集的复杂程度的增加，产生的概率越大。

但是如果在实际的归结操作中，我们往往关注的是我们的所给的**子句集是否能够归结出矛盾**也就是空子句，从而证明我们的结论，那么深度优先的归结策略可以更加高效的得出结论。

四、 参考资料

[1] On the logical skills of large language models: evaluations using arbitrarily complex first - order logic problems [arXiv preprint]. arXiv:2502.14180.

[2] https://github.com/yizuodi/SYSU2022_AI/blob/main/Experiment_2/code/main.py